Massively Parallel Data Processing on Infrastructure as a Service Platforms

vorgelegt von
Dipl.-Inf. Daniel Warneke
aus Berlin

der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
- Dr. rer. nat. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Sahin Albayrak
Gutachter:	Prof. Dr. Odej Kao
	Prof. Dr. Volker Markl
	Prof. Dr. Felix Naumann

Tag der wissenschaftlichen Aussprache: 28. September 2011

Berlin 2011
D 83

Acknowledgement

I would like to take the chance of expressing my sincere gratitude to a number of people who accompanied me in the course of my studies and helped shaping this thesis.

First and foremost I would like to thank my advisor Odej Kao. He provided me with a great working environment in his research group and supported my work through many helpful comments and collaboration opportunities. I also would like to express my appreciation to Volker Markl and Felix Naumann for agreeing to review this thesis.

Much of the great work atmosphere I experienced at TU Berlin is owed to the many people I had the opportunity to work with. First of all, I would like give credit to the wonderful members of the CIT team (past and present), especially Dominic Battré, Philipp Berndt, Björn Lohrmann, André Höing, Matthias Hovestadt, and Martin Raack. They were always open to discussions (on research and non-research related matters) and provided help on technical problems whenever possible. Moreover, I greatly appreciated their company during the countless evenings we spent working late together in the office.

I would also like to thank the many master students (and partly now colleagues) I was lucky enough to work with, namely Natalia Frejnik, Andreas Kliem, Alexander Stanik, and Mareike Strüwing. They all helped to develop many valuable ideas of this thesis and contributed to the prototypic implementation. Siddhant Goel was a summer intern at our research group in 2009 and helped building the foundation for the topology inference research which is presented in this thesis.

Furthermore, I must not forget to express my gratitude to the DIMA research group of TU Berlin, in particular Volker Markl, Fabian Hüske, and Stephan Ewen. Volker was kind enough to take me with this group on several research trips and introduced me to many interesting people of the database community. Fabian and Stephan provided very helpful feedback on the design of the Nephele framework and were always open to discussions on new ideas centering around large-scale data analysis systems.

Finally, I would like to thank my dear family for their love and support, especially my father Joachim who proofread this thesis, although computer science is most certainly not his favorite subject.

Abstract

In recent years, Infrastructure as a Service (IaaS) clouds have emerged as a promising new platform for massively parallel data processing. By eliminating the need for large upfront capital expenses, operators of IaaS clouds offer their customers the unprecedented possibility to acquire access to a highly scalable pool of computing resources on a short-term basis and enable them to execute data analysis applications at a scale which has been traditionally reserved to large Internet companies and research facilities.

However, despite the growing popularity of these kinds of distributed applications, the current parallel data processing frameworks, which support the creation and execution of large-scale data analysis jobs, still stem from the era of dedicated, static compute clusters and have disregarded the particular characteristics of IaaS platforms so far.

This thesis revisits the design of a parallel data processing framework against the background of the new possibilities and challenges of IaaS clouds with the objective of improving the processing efficiency on these platforms in terms of both time and cost. In particular, the thesis analyzes how parallel data processing frameworks can take advantage of the cloud's ability for rapid resource provisioning and presents a new parallel data processing framework called Nephele, which explicitly exploits these new cloud features. Moreover, several approaches are presented to reduce the increased risk of I/O bottlenecks during the job execution which results from the cloud's use of hardware virtualization.

In order to underline their effectiveness, all contributions of this thesis are evaluated through various practical experiments and, whenever possible, contrasted to the state of the art in the respective field.

Zusammenfassung

Infrastructure as a Service (IaaS) Clouds haben sich in den vergangenen Jahren zu einer vielversprechenden neuen Plattform für massiv-parallele Datenverarbeitung entwickelt. Durch den Wegfall der Notwendigkeit hoher Anfangsinvestitionen bieten Betreiber von IaaS Clouds ihre Kunden die nie dagewesene Möglichkeit, kurzzeitigen Zugriff auf einen hoch skalierbaren Pool von Rechenressourcen zu erhalten und darauf Datenanalyseprogramme in einer Größenordnung auszuführen, die bislang nur großen Internetfirmen und Forschungseinrichtungen vorbehalten war.

Trotz der steigenden Popularität dieser Form von verteilten Anwendungen, stammen die aktuellen Datenverarbeitungsframeworks, die die Erstellung und Ausführung dieser großangelegten Aufgaben (Jobs) zur Datenanalyse unterstützen, immernoch aus der Ära der dedizierten, statischen Rechencluster und haben die speziellen Eigenschaften der IaaS Plattformen bislang außer Acht gelassen.

Diese Doktorarbeit greift den Entwurf eines parallelen Datenverarbeitungsframeworks vor dem Hintergrund der neuen Möglichkeiten und Herausforderungen einer IaaS Cloud neu auf, und zwar mit dem Ziel, die Verarbeitungseffizienz von Jobs auf dieser Plattform sowohl in Hinblick auf die Zeit als auch auf die Kosten zu verbessern. Dabei analysiert die Arbeit, wie ein Framework für parallele Datenverarbeitung die Fähigkeiten einer Cloud zur schnellen Ressourcenbereitstellung nutzen kann und präsentiert daraufhin ein neues Verarbeitungsframework mit dem Namen Nephele, welches diese neuen Möglichkeiten der Cloud explizit ausnutzt. Darüber hinaus werden noch mehrere Ansätze zur Reduzierung des erhöhten Risikos von I/O Flaschenhälsen während der Jobausführung vorgestellt, welches in einer Cloud durch die Verwendung von Hardwarevirtualisierung entsteht.

Um ihre Leistungsfähigkeit aufzuzeigen, werden alle Beiträge dieser Doktorarbeit durch zahlreiche praktische Experimente evaluiert und, sofern möglich, mit dem aktuellen Stand der Technik gegenübergestellt.

Contents

1. Introduction **1**
1.1. Problem Definition . . . 3
1.2. Contribution . . . 5
1.3. Outline of the Thesis . . . 7

2. Characteristics of Infrastructure as a Service Clouds **9**
2.1. Service Models of IaaS Clouds . . . 9
2.1.1. Compute Service Models . . . 10
2.1.2. Storage Service Models . . . 11
2.1.3. Service Level Agreements . . . 13
2.2. User Interface to IaaS Clouds . . . 13
2.3. Performance Characteristics . . . 15
2.3.1. CPU Performance Characteristics . . . 16
2.3.2. I/O Performance Characteristics . . . 19
2.4. Summary . . . 24

3. Exploiting Dynamic Resource Allocation **27**
3.1. Design Principles . . . 28
3.2. The Nephele Parallel Data Processing Framework . . . 29
3.2.1. Architecture . . . 29
3.2.2. Job Description . . . 31
3.2.3. Job Scheduling and Execution . . . 33
3.3. Parallelization and Scheduling Strategies . . . 36
3.3.1. Finding Suitable Degrees of Parallelism and VM Types . . . 36
3.3.2. Automatic VM Allocation and Deallocation . . . 38
3.4. Evaluation . . . 39
3.4.1. Experiment 1: MapReduce and Hadoop . . . 40
3.4.2. Experiment 2: MapReduce and Nephele . . . 41
3.4.3. Experiment 3: DAG and Nephele . . . 44
3.4.4. Results . . . 46
3.5. Related Work . . . 50
3.6. Summary . . . 52

4. Detecting Bottlenecks in Parallel Data Flow Programs **53**
4.1. Processing Model and Problem Definition . . . 54
4.1.1. Processing Model . . . 55
4.1.2. Problem Definition . . . 56
4.2. Bottleneck Detection Algorithms . . . 57
4.3. Implementation in Nephele . . . 60
4.4. Evaluation . . . 62
4.4.1. Use Case . . . 63
4.4.2. Results . . . 64
4.5. Related Work . . . 67
4.6. Summary . . . 69

5. Mitigating I/O Variations with Adaptive Compression **71**
5.1. Design Principles . . . 72
5.2. Adaptive Online Compression in IaaS Clouds . . . 74
5.2.1. Decision Model . . . 74
5.2.2. Implementation in Nephele . . . 76
5.3. Evaluation . . . 79
5.3.1. Adaptivity . . . 79
5.3.2. Changing Data Compressibility . . . 82
5.4. Related Work . . . 83
5.5. Summary . . . 84

6. Topology Inference in IaaS Clouds **87**
6.1. Analysis of Network Path Characteristics in Clouds . . . 89
6.1.1. Inference based on Packet Loss . . . 90
6.1.2. Inference based on Packet Delay . . . 92
6.1.3. Discussion . . . 94
6.2. Examining the Accuracy of Topology Inference . . . 95
6.2.1. Obtaining Initial Similarity Values for the VMs . . . 95
6.2.2. Accuracy of the Inferred Topologies . . . 96
6.2.3. Transferring Binary Trees into General Trees . . . 99
6.3. Implementation in Nephele . . . 100
6.4. Evaluation . . . 102
6.5. Related Work . . . 103
6.6. Summary . . . 105

7. Interaction with Higher Layer Components **109**
7.1. The Stratosphere Software Stack . . . 109
7.2. The PACT Layer . . . 112
7.2.1. The Structure of a PACT Program . . . 113

7.2.2. The PACT Input Contracts 114
7.2.3. The PACT Output Contracts 117
7.2.4. Running PACT Programs on Nephele 118
7.3. Optimization Opportunities through the PACT Layer 120
7.4. Summary 122

8. Conclusion **125**

A. Appendix **129**

1. Introduction

Contents

1.1. Problem Definition 3
1.2. Contribution 5
1.3. Outline of the Thesis 7

Today a growing number of companies have to process data sets which are increasing exponentially in complexity and size. Classic representatives of these companies are operators of Internet search engines, like Google, Yahoo, or Microsoft, which have to crawl the web for information and prepare it for efficient retrieval. However, besides this traditional web search use case, new use cases have constituted a growing demand for large-scale data analysis in both industry and academia.

Examples of these use cases are manifold: They range from large scientific simulations in the scale of tera or even peta bytes [63], over the processing of radio-frequency identification (RFID) data in the domain of chain supply management [60] to data analysis problems from the field of life sciences [68].

The challenge of processing these large range of data sets in a scalable and cost-efficient manner has rendered traditional database solutions prohibitively expensive [37]. Instead, recent price trends for commodity hardware and multicore CPUs have popularized an architectural paradigm which favors a massively parallel data processing on a large set of inexpensive computers over expensive servers. Problems like processing crawled documents or regenerating a web index are split into several independent subtasks, distributed among the available nodes, and computed in parallel.

In order to simplify the development of distributed applications on top of such highly parallelized architectures, many Internet companies have started to build customized data processing frameworks. Examples are Google's MapReduce [43], Microsoft's Dryad [74], or Yahoo!'s Map-Reduce-Merge [147]. They can be classified by terms like high throughput computing (HTC) or many-task computing (MTC), depending on the amount of data and the number of tasks involved in the computation [106]. Parts of the data processing community also use the term data-intensive scalable computing (DISC) frameworks to refer to them [52]. Although these systems differ in design, their programming models share similar objectives, namely hiding the hassle of parallel programming, fault

tolerance, and execution optimizations from the developer. Developers can typically continue to write sequential programs. The processing framework then takes care of distributing the program among the available nodes and executes each instance of the program on the appropriate fragment of data.

However, despite the price decline for commodity hardware, a reasonable application of such parallel data processing frameworks has traditionally been limited to companies which have specialized in large-scale data analysis and therefore operate their own data centers. For other institutions, which only have to carry out such analysis tasks occasionally, the cost for acquiring and maintaining a large IT infrastructure has often been prohibitive. Parts of the computer science community have tackled this problem with grid computing [55]. Its overall idea is to combine servers from multiple administrative domains to a shared resource pool which can temporarily be dedicated to specific applications. Although the developed protocols did not lead to a software environment that grew beyond its community [20], grid computing was driven by the vision of transforming compute and storage resources from something that one buys and operates oneself to something that is operated by a third party [56] and can be consumed as a service.

A similar vision has recently gained wide-spread attention under the term *cloud computing*. Cloud computing refers to the idea of delivering dynamically-scalable IT resources like computing power, storage or higher level platforms and services on demand to external customers over the Internet [56]. The resources are rapidly provisioned and released with minimal management effort or service provider interaction [93], allowing the cloud customer to quickly grow or shrink the rented infrastructure according to his requirements without long term commitment or the need of large capital expenses.

Operators of so-called Infrastructure as a Service (IaaS) clouds, like Amazon Elastic Compute Coud (EC2) [8] or Rackspace [105], have specialized in offering their customers access to storage and computing resources hosted within their data centers. The size of their data centers typically adds up to thousands or ten thousands of computers, creating the impression of virtually unlimited resources to the customer. To facilitate their flexible allocation, the computing resources are typically deployed in form of virtual machines (VMs). The usage of the leased resources is charged on a short-term basis (for example, processors by the hour and storage by the day), thereby rewarding conservation by letting machines and storage go when they are no longer useful [20].

The prospect of using a thousand VMs for one hour at the same cost as one VM for a thousand hours has made IaaS clouds an attractive new platform for the above-mentioned parallel data processing frameworks. In 2007, the New York Times announced they used 100 VMs of the Amazon EC2 cloud in parallel to convert large parts of their article archive into publicly available PDF files in under 24 hours [61]. Projects like Hadoop [124], a popular open source implementation of Google's MapReduce framework, have already begun to promote using their frameworks in the cloud [141]. As a

result of the growing interest, Amazon integrated Hadoop as one of its core infrastructure services [14] in 2009. Since then, several companies have become known as customers of this parallel data processing platform (e.g. [16, 19]).

1.1. Problem Definition

Although the VM abstraction of IaaS clouds fits the architectural paradigm assumed by the current data processing frameworks, it is important to recall that these data processing frameworks have been designed with traditional, dedicated data centers in mind. These dedicated data centers are essentially static in size and fully controlled by that party which also runs the data processing framework. The key to an IaaS cloud's economic operation, however, lies in statistical multiplexing [20]. This means many different parties dynamically share the same physical hardware for a specific period of time, but are separated from each other through hardware virtualization.

Compared to dedicated data centers, the statistical multiplexing allows for a much more flexible and cost-efficient access to the required computing resources for the individual cloud customers on the one hand. On the other hand, the necessary virtualization also results in a loss of control over the physical hardware environment. With respect to parallel data processing this leads to new opportunities but also challenges:

The biggest *opportunity* for parallel data processing on IaaS platforms is probably the rapid resource provisioning. In contrast to classic cluster setups, new VMs can be allocated at any time through a well-defined interface and become available in a matter of seconds. Machines which are no longer used can be terminated instantly and the cloud customer will not be charged for them any longer. Moreover, compared to the traditional data center setups, most IaaS clouds offer VMs with different hardware characteristics (number of CPU cores, amount of main memory, etc.) and at different cost, providing the possibility of heterogeneous computing resources when needed.

However, instead of embracing the dynamic resource allocation, current data processing frameworks rather expect the cloud to imitate the static nature of the cluster environments they were originally designed for. For example, Amazon Elastic MapReduce [14], currently a major product for cloud data processing, does not support to change the set of allocated VMs for a processing job in the course of its execution, although the tasks the job consists of might have completely different demands on the environment. As a result, the rented resources may be inadequate for big parts of the processing job, which may lower the overall processing performance and increase the cost.

In contrast to that, a data processing framework which exploits both the rapid resource provisioning and the possible resource heterogeneity could flexibly adapt the rented

computing resources to the demands of the processing job. That way a processing job would no longer have to be executed on the same static set of compute nodes it was originally started on. Instead, following the idea of a cloud, the required resources could be allocated *on demand* according to the job's current processing phase and adequate to the hardware demands of the job's individual tasks. After each phase, the machines could be released and no longer contribute to the overall processing cost.

Facilitating such use cases entails several fundamental changes in the design of a data processing framework which have not been investigated so far. For example, compute nodes can no longer be assumed to be *owned* by the data processing framework. Instead, the framework must be able to *lease* the required resources from the cloud with a clear notion of monetary cost. The way to model this leasing, from the programming abstraction down to the resource scheduler, is currently an open research question. Moreover, there exist no allocation strategies for VMs which aim at optimizing the processing cost of individual MTC-like applications.

Besides the new opportunities, IaaS clouds also lead to new *challenges* which also must be taken into account for efficient parallel data processing. As indicated above, one major challenge pertains to the loss of control over the physical hardware compared to classic cluster setups.

This loss of hardware control carries particular weight with respect to the available I/O bandwidth. Since the VMs of different customers may be colocated on the same host, the I/O workloads induced by one VM can affect the I/O performance of the other machine negatively [71]. This can lead to unpredictable performance fluctuations which render large parts of the rented computing resources underutilized. Moreover, current data processing frameworks offer to take the network topology, i.e., the way the computing resource are physically connected to each other, into account in order to exploit data locality [43]. In an IaaS cloud, however, the physical network topology is typically not exposed to the customer, which further increases the risk of I/O bottlenecks. If and to what extent parallel data processing frameworks or cloud applications in general can counteract these limitations is still an open question in the research community.

Another research question that arises from the cloud's ability to provide large sets of compute nodes on demand is how to assist developers in finding reasonable degrees of parallelization for their processing jobs. Since there is no longer a predetermined limit on the number of available resources like in a classic cluster setup, developers might be tempted to reduce the completion times of their jobs by larger and larger scale-outs. Of course, the vast majority of jobs cannot be parallelized indefinitely, but will experience insuperable I/O bottlenecks at some point. However, the problem of determining this very point in an environment with virtually unlimited computing power has not been addressed so far.

In sum, current frameworks for massively parallel data processing have disregarded both the positive and negative implications resulting from the characteristics of IaaS clouds so far. Neither can these frameworks exploit the cloud's new abilities for dynamic resource allocation, nor do they include mechanisms to cover up for the loss of control over the hardware environment or traditional parallelization guidelines. As a result of the status quo, the problem addressed by this thesis can be summarized as

> "With regard to the characteristics of IaaS clouds, how can one improve the efficiency of massively parallel data processing on these new platforms?"

Due to the prevalent cost models of today's IaaS clouds, the term efficiency can be understood in two ways. On the one hand, it may refer to the *processing time*, i.e., the period of time a cloud customer must wait for his data processing job to complete. On the other hand, efficiency may also refer to the *processing cost*, i.e., the amount of money the cloud customer has to pay to the cloud operator after the completion of his processing job.

1.2. Contribution

The contributions of this thesis with respect to massively parallel data processing on IaaS platforms can be found in two major areas, *the exploitation of the cloud's dynamic resource allocation* and *the mitigation of the loss of control over the hardware environment* compared to classic cluster setups.

In the first area, the thesis presents the first data processing framework to explicitly exploit the flexible resource provisioning offered by today's IaaS. The new framework named *Nephele* offers to flexibly adjust the number of allocated VMs according to the current processing workload. In the course of a processing job the required VMs are automatically instantiated and terminated. Moreover, Nephele offers support for *heterogeneous* compute nodes, i.e., different tasks of a processing job can be assigned to different types of VMs according to their individual demand for CPU power, main memory, etc. Based on the characteristics of today's IaaS clouds, the thesis describes Nephele's fundamental design principles, discusses different strategies for the allocation of VM, and highlights the resulting performance benefits through a comparison with the popular Hadoop framework. Moreover, it presents a novel approach to detect bottlenecks in parallel data flow programs in order to alleviate the problem of missing parallelization guidelines for data processing in the cloud.

In the second area, this thesis introduces two new approaches to counteract the loss of hardware control as described in the previous section. The first approach makes use of *adaptive online compression* to respond to changes in the VMs' available I/O bandwidth.

The second approach borrows from the field of network topology inference [98]. It uses *end-to-end measurements* to gain knowledge about the cloud's physical network topology and exploit it in order to improve the processing job's data locality. Unlike existing work in the respective fields, both approaches have been explicitly designed towards the characteristics of IaaS clouds. Especially the impact of different hardware virtualization techniques has been carefully studied and considered in the design process. As shown through various experimental evaluations, both approaches can improve the I/O performance of parallel data processing on IaaS platforms significantly.

Parts of this thesis have been published in the following publications:

Journals

1. **Daniel Warneke, Odej Kao**
   Exploiting Dynamic Resource Allocation for Efficient Parallel Data Processing in the Cloud
   In: IEEE Transactions on Parallel and Distributed Systems, 22(6), pp. 985-997, 2011

2. **Alexander Alexandrov, Dominic Battré, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, Daniel Warneke**
   Massively Parallel Data Analysis with PACTs on Nephele
   In: Proceedings of the VLDB Endowment, 3(1-2), pp. 1625-1628, 2010

Proceedings

3. **Daniel Warneke, Odej Kao**
   Nephele: Efficient Parallel Data Processing in the Cloud
   In: Proceedings of 2nd Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS '09), pp. 1-10, 2009

4. **Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, Daniel Warneke**
   Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing
   In: Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10), pp. 119-130, 2010

5. **Dominic Battré, Matthias Hovestadt, Björn Lohrmann, Alexander Stanik, Daniel Warneke**
   Detecting Bottlenecks in Parallel DAG-based Data Flow Programs

In: Proceedings of 3rd Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS '10), pp. 1-10, 2010

6. **Alexander Alexandrov, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, Daniel Warneke**
   MapReduce and PACT - Comparing Data Parallel Programming Models
   In: Proceedings of the 14th Conference on Database Systems for Business, Technology, and Web (BTW 2011), pp. 25-44, 2011

7. **Matthias Hovestadt, Odej Kao, Andreas Kliem, Daniel Warneke**
   Evaluating Adaptive Compression to Mitigate the Effects of Shared I/O in Clouds
   In: Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW '11), pp. 1042-1051, 2011

8. **Dominic Battré, Natalia Frejnik, Siddhant Goel, Odej Kao, Daniel Warneke**
   Inferring Network Topologies in Infrastructure as a Service Clouds
   In: Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGRID '11), pp. 604-605, 2011

9. **Dominic Battré, Natalia Frejnik, Siddhant Goel, Odej Kao, Daniel Warneke**
   Evaluation of Network Topology Inference in Opaque Compute Clouds Through End-to-End Measurements
   In: Proceedings of the 4th IEEE International Conference on Cloud Computing (CLOUD '11), pp. 17-24, 2011

1.3. Outline of the Thesis

The remainder of this thesis is structured as follows:

Chapter 2: Characteristics of Infrastructure as a Service Clouds

Chapter 2 highlights the currently prevalent service models of IaaS clouds, describes the most common ways a customer can interact with these systems, and elaborates on typical performance characteristics of these platforms.

Chapter 3: Exploiting Dynamic Resource Allocation

Chapter 3 discusses the fundamental design principles a parallel data processing framework must meet in order to facilitate dynamic resource allocation.

Based on these design principles, the chapter presents the new Nephele framework and demonstrates possible efficiency gains through the integration of the new cloud capabilities in the job processing.

Chapter 4: Detecting Bottlenecks in Parallel Data Flow Programs

Chapter 4 deals with the problem of finding reasonable scale-outs for parallel processing jobs in IaaS environments. In this context, the chapter presents a novel scheme for the detection of CPU and I/O bottlenecks in these setups and demonstrates the integration in Nephele.

Chapter 5: Mitigating I/O Variations with Adaptive Compression

Chapter 5 discusses the problem of I/O variations as a result colocated VMs in IaaS clouds and presents a novel adaptive compression scheme to mitigate these effects.

Chapter 6: Topology Inference in IaaS Clouds

Chapter 6 addresses the lack of topology information for efficient parallel data processing in clouds. In order to deal with this problem, the chapter evaluates topology inference based on end-to-end measurements and presents an improved inference approach to reconstruct likely data center network structures with or without the assistance of internal network components.

Chapter 7: Interaction with Higher Layer Components

Chapter 7 considers the contributions of this thesis in the larger scope of the Stratosphere research project and discusses optimization opportunities with regard to semantically richer layers of the Stratosphere software stack.

Chapter 8: Conclusion

Chapter 8 concludes the thesis with a summary of the contributions and an outlook on promising future research topics.

2. Characteristics of Infrastructure as a Service Clouds

Contents

2.1. Service Models of IaaS Clouds **9**
2.1.1. Compute Service Models 10
2.1.2. Storage Service Models 11
2.1.3. Service Level Agreements 13
2.2. User Interface to IaaS Clouds **13**
2.3. Performance Characteristics **15**
2.3.1. CPU Performance Characteristics 16
2.3.2. I/O Performance Characteristics 19
2.4. Summary **24**

Following the mission statement of this thesis, this chapter will introduce the anatomy of an IaaS cloud and contrast its economic and technical properties against those of traditional data centers from a user's perspective. The basis for the examination will in many parts be the Amazon EC2 cloud [8], which is widely regarded as a de-facto standard for IaaS platforms [64].

2.1. Service Models of IaaS Clouds

The overall service model of IaaS clouds centers around the provisioning of flexible and scalable IT infrastructures over the Internet. Customers of IaaS clouds can dynamically adapt the size of the leased infrastructure to their demands, thereby shifting the classic data center risks of over- or underprovisioning to the cloud operator [20].

In general, IaaS clouds distinguish between compute services, i.e., the provisioning of VMs, and storage services. This section will exemplarily describe both service models by means of the popular IaaS platform Amazon EC2. Other IaaS providers like Rackspace [105] or GoGrid [58] have implemented comparable service models with only minor deviations.

2.1.1. Compute Service Models

The Amazon Elastic Compute Coud (EC2) represents the compute service component of Amazon's overall cloud computing platform Amazon Web Services (AWS) [10]. According to its service model, customers of Amazon EC2 can rent computing power from Amazon's data centers on a pay-per-usage basis. Following the idea of Infrastructure as a Service, the computing power is provided in the form of VMs which can be dynamically created or destroyed and fully customized by the customer.

A VM, also called *instance* in Amazon's terminology, is always created from a disk image which must be specified by the customer. The disk image acts as a template for the new VM's initial disk content. In the most basic case, the disk image only contains the VM's guest operating system, however, it may also contain additional software components. Customers of Amazon EC2 can either provide custom disk images or choose from a set of public images which typically include free software whose license agreements permit an unrestricted distribution, such as GNU/Linux.

Besides the disk image, a customer can also specify the type of VM to be started within the cloud operator's data center, the so-called *instance type*. The instance type describes the hardware characteristics of the rented compute nodes. Each instance type refers to a distinct combination of processing power, amount of main memory, and disk space. Moreover, the instance type also influences the price of the VM. Amazon offers a fixed set of different instances types as shown in Table 2.1.

Name	Computing Power	Main Memory	Storage	Price per Hour
Small	1 Compute Unit	1.7 GB	160 GB	0.10 $
Large	4 Compute Units	7.5 GB	850 GB	0.40 $
Extra Large	8 Compute Units	15 GB	1690 GB	0.80 $
High-CPU Medium	5 Compute Units	1.7 GB	350 GB	0.20 $
High-CPU Extra Large	20 Compute Units	7 GB	1690 GB	1.20 $

Table 2.1.: Overview of Amazon EC2's available VM types (instance types) and prices as of September 2009 [8].

Amazon expresses the computing power of each VM type in a custom unit, the so-called *Amazon EC2 Compute Unit*. According to Amazon's website [8], an Amazon EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. Amazon has chosen to introduce a custom definition to better manage the consistency and predictability of compute capacity across different (generations of) hardware platforms.

Besides the type, the price a customer has to pay for a VM depends on two additional factors: The first factor is the period of time the machine has been running. Here,

Amazon has popularized the per-hour pricing model, i.e., a customer is charged for the VM usage by the hour with partial hours being billed as full hours. After being shutdown, the VM is destroyed and incurs no additional charges. The disk content of destroyed VMs is typically discarded unless the customer has chosen to create a snapshot of the VM before. In this case, the machine can be restarted from the snapshot, however at the expense of a small fee which Amazon charges for storing the snapshot.

Recently, Amazon extended its initial per-hour pricing model by two refinements. Since Amazon EC2 generally does not guarantee that a customer's request for VMs can always be fulfilled, they introduced so-called *reserved instances* besides the regular instances. With reserved instances customers can make a one-time payment for each VM they want to reserve for a particular time span, for example one year. In turn, Amazon reserves the physical resources required to run the VM during that time span and allows a discount on the hourly usage charge.

The latest refinement to Amazon's per-hour price model has been introduced with the so-called *spot instances*. Amazon uses spot instances to sell temporary excess capacity in their data centers. A customer can bid a price he is willing to pay for a particular instance type per hour. Depending on the current demand for computing resources, Amazon launches the requested VM at the bidden price. Spot instances are often significantly cheaper than regular instances [149]. However, in case the other customers agree to pay a higher hourly fee, Amazon reserves the right to shut down already started spot instances without further notice in order to regain physical resources.

The second major cost factor of a VM is the network traffic it causes. While internal network traffic, i.e., network traffic among instances inside the same data center, is free of charge, data transfers to other EC2 data centers or, in general, other hosts on the Internet come at an additional fee. The concrete cost per GB of data depends on the overall amount of data sent within a month and ranged between 0.10 and 0.17 $ per GB as of September 2009 [8].

2.1.2. Storage Service Models

As indicated in the previous subsection, the flexible per-hour pricing model of most IaaS clouds leads to ephemeral storage inside the VMs which is discarded after the respective machine is shut down. In order to provide persistent storage, i.e., storage that is preserved beyond the lifetime of a VM, many operators of IaaS clouds complement their compute services with additional storage services. The usage of these services is usually billed according to separate pricing model, so it can be offered independently of the compute nodes' lifetimes.

In the context of IaaS, Amazon offers two noteworthy components for persistent storage in their AWS portfolio, *Amazon Elastic Block Storage (EBS)* [12] and *Amazon Simple Storage Service (S3)* [9].

Amazon EBS has been explicitly designed to work with the EC2 compute nodes. A customer can create different EBS volumes of up to one TB in size and attach each volume to a particular EC2 instance which is hosted in the same data center. To the EC2 instance the EBS volume looks like a regular block device, which can be formatted with a file system and then accessed through the operating system's regular file system primitives. Amazon charges customers for each EBS volume on a monthly basis, ending with the month after the respective volume has been deallocated. As of September 2009, the cost of one GB of storage on EBS was 0.10 $ per month [12]. Moreover, Amazon demanded an additional fee of 0.10 $ for every one million I/O requests issued by an EC2 instance to an EBS volume.

In contrast to Amazon EBS, Amazon S3 can be used independently of the compute service. According to its website [9], Amazon S3 has been intentionally built with a minimal feature set but with a strong focus on high scalability and reliability.

Amazon S3 considers the data to be stored as *objects* which are organized into one or more so-called *buckets*. The size of an object can range from one byte to five GB. The number of objects that can be stored is unlimited. Unlike Amazon EBS, S3 storage cannot directly be attached to a compute instance as a raw block device. Instead, the access and manipulation of the stored data takes place over HTTP with either the SOAP [65] or REST [53] protocol. Moreover, Amazon S3 allows customers to concurrently access the stored data from multiple locations. Concurrent modifications to the redundantly stored objects are resolved by a protocol providing eventual consistency [85].

The usage of S3 storage is also billed on a monthly basis. The exact monthly fee is composed of the pure storage cost, cost for incoming and outgoing data transfers as well as cost for accessing and modifying objects. Moreover, the service prices vary across different geographical regions. As of September 2009, Amazon charged users of their US-based data centers a monthly fee between 0.12 and 0.14 $ per GB of storage, between 0.10 and 0.17 $ per GB of outgoing data transfer, and 0.10 $ per GB of incoming data transfer [9]. In addition, 1000 HTTP PUT, COPY, POST, or LIST requests were billed with 0.01 $, as well as 10000 HTTP GET requests.

The cost for data transfer only applies when data packets are actually sent over the Internet. Data transfer within the same geographic location is free of charge. In particular, VMs hosted on Amazon EC2 can access the storage service at no cost for data transfer within the same data center.

For large data transfers, Amazon also offers customers to send in physical storage devices like hard drives and therefore avoid excessive data transfer cost [11]. The content of the

device is then either copied to the S3 storage, or vice versa, a backup of the S3 data is created on the physical storage device and then sent back to the customer. The cost of the service was 80 $ per physical storage device as well as 2.49 $ per data-loading hour as of 2009. Moreover, Amazon's regular fees for HTTP requests to the S3 storage add to the overall cost.

2.1.3. Service Level Agreements

A service level agreement (SLA) is a negotiated agreement between a customer and a service provider. As part of a legal contract, it defines a common understanding about the offered service, rights and duties of involved parties, as well as consequences of violating the agreement. Many operators of IaaS clouds (e.g. [8, 105, 58]) also offer SLAs to their customers in order to underline the reliability of their services.

However, in the context of IaaS clouds, it is important to notice that current SLAs mostly cover the *availability* of the offered service and provide very few guarantees, if at all, about its *quality*. For example, Amazon EC2 commits itself to an annual uptime of at least 99.95% per year and VM [7]. In the event Amazon does not meet this commitment, the affected customers can claim a refund on their monthly bill. Any legally binding statements with regard to the available compute performance or I/O throughput are not part of the SLA.

The company GoGrid [58] is one of the few IaaS providers which has included concrete performance figures in its SLA [59]. Besides the regular availability commitment, the document defines upper bounds on the network latency and packet loss their customers have to tolerate. However, any guarantees about the CPU performance or network throughput are also excluded from the agreement.

2.2. User Interface to IaaS Clouds

According to the definition of the National Institute of Standards and Technology (NIST), cloud computing is highlighted by the customers' ability to rapidly provision and release computing resources with minimal management effort or service provider interaction [93]. Following this idea, IaaS providers have implemented a variety of interfaces which their customers can use to manage the rented compute and storage resources. Figure 2.1 summarizes the most common interfaces to an IaaS cloud from the perspective of the cloud customer.

One interface that all IaaS providers have in common is a management interface which their customers can use to allocate, release, and manage both their computing as well

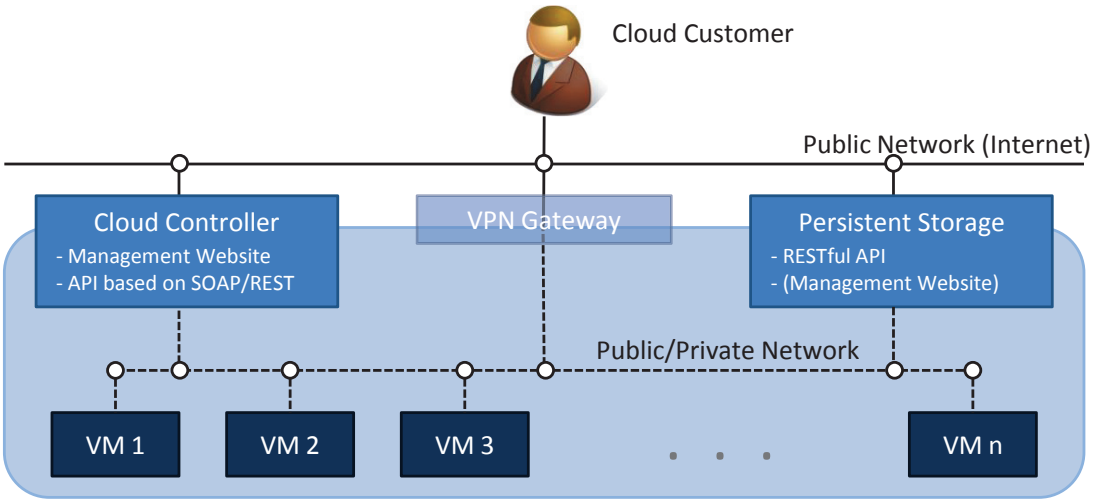


Figure 2.1.: Common interfaces to an IaaS cloud from a customer's perspective.

as storage resources. For the remaining chapters this thesis will refer to this interface as the *Cloud Controller* interface. The location of this Cloud Controller interface, i.e., its URL, must be well-known to the cloud customer. Moreover, the interface is in general accessible from a public network such as the Internet.

Typically, the Cloud Controller interface allows the customer to issue management requests in two ways. The first way is through a classic website to which the customer can log on and overview his deployment, such as Amazon's AWS Management Console [17]. The second way is through a remote prodecure call (RPC) API. This RPC API enables the customer to adapt his current cloud setup from external applications. For example, Amazon EC2 exposes all functions required to manage a deployment on their EC2 cloud through a well-defined Web-Service interface [13]. As a result, a plethora of tools has evolved around this cloud platform in recent years, offering to simplify management tasks and start or stop VMs on behalf of the customer. The authenticity of each RPC call is ensured by a combination of certificates and secret keys which Amazon issues to the respective customer upon registration.

After the customer has started a VM, either through the management website or the RPC interface, the machine is instantiated from the selected disk image and booted on a suitable host inside the cloud operator's data center. According to recent evaluations, the average startup time for VMs on Amazon EC2, i.e., the time from issuing the creation request until the machine switches to the ready state, ranges from approximately one to three minutes [46, 92, 102]. During the boot process, the VM is assigned a network address. The customer learns about the VM's network address through the Cloud

Controller interface and can then use it to establish a connection to the machine.

The nature of the assigned network address may depend on the concrete cloud provider. For example, Amazon EC2 assigns each started VM a public IP address from their pool of available network addresses. As a result, each cloud customer has the possibility to access the rented machines directly over the Internet. Other cloud providers, like Zimory [152], may also assign the launched VMs non-public IP addresses, so accessing the VMs from the Internet may involve a *virtual private network (VPN) gateway*. In either case, however, the communication among the allocated VMs of a customer within the cloud is typically possible without any restriction if not explicitly configured.

The third most common interface besides the Cloud Controller and the possible VPN gateway is the interface to manage the cloud's *persistent storage* service. The persistent storage service is usually accessible from both the Internet as well as any private network which may be deployed among the customer's VMs. Similar to the Cloud Controller interface, all functions required to manage and access the stored data are typically exposed through a public API. Many cloud providers [9, 58, 105] have established lightweight access and transfer protocols based on the Representational State Transfer (REST) principles [53]. By also building upon HTTP as the transport protocol, these RESTful APIs have a similar robustness compared to classic Web-Service protocols like SOAP, but are widely considered to have a lower processing complexity [4, 96]. In addition to the API, many IaaS providers also offer a dedicated website to manage and access the persistent storage (e.g. [17]).

2.3. Performance Characteristics

After having explained the economic properties of an IaaS cloud, the thesis will now discuss typical performance characteristics of this new computing platform. The characteristics have been gathered from previous performance studies and independent microbenchmarks. Since there is no standardized technology stack for building an IaaS cloud, the thesis will again focus on the currently most popular cloud system Amazon EC2. However, in order to provide a broader picture, this section will also investigate the influence of virtualization in general on the CPU and I/O performance. These virtualization benchmarks have been conducted on our local IaaS testbed running the Eucalyptus [99] cloud software. A thorough description on the setup can be found in the appendix.

2.3.1. CPU Performance Characteristics

The CPU performance characteristics that can be observed in IaaS clouds or virtualized environments in general depend heavily on the considered class of application.

For CPU-intensive applications, like classic high-performance computing (HPC) applications, the performance impact of virtualization has traditionally been low [23]. The reason for this is that CPU-intensive applications predominantly run user level code which can be executed natively on the physical processor. So-called privileged instructions, which may compromise the hypervisor's control over the physical hardware and therefore must be handled separately, are only called seldom in this domain [70]. Moreover, modern CPUs feature special virtualization extensions to reduce the overhead of these calls and improve memory management across different VMs [132]. As a result, CPU-intensive applications running inside a VM today typically achieve a comparable performance to native systems [145].

For data-intensive applications, which represent the focus of this thesis, the situation is different. Since every I/O operation leads to at least one privileged instruction, the hypervisor may spend considerable amounts of time translating these calls so that they comply with the x86 privilege level architecture.

Although the CPU overhead for virtualizing I/O operations has been addressed by several researchers in recent years [94, 39, 144, 127], the concrete performance impact is still hard to quantify in general. It depends on the type of virtualization layer (e.g. full virtualization or paravirtualization), the type of I/O requests (network or disk I/O), and their frequencies. Moreover, ongoing improvements in the implementation of the individual virtualization layer may also influence the amount of overhead.

Further complicating matters, the CPU overhead for virtualizing I/O operations is even hard to assess for an application inside a VM at runtime because the displayed CPU utilization often does not reflect the overhead correctly. In order to illustrate both the varying CPU overhead as well as the inaccurate display of the CPU utilization inside the VMs, we conducted a series of experiments on our local IaaS testbed [69]. In the scope of these experiments, we created set of small auxiliary programs to generate network and disk I/O load. Then we contrasted the displayed CPU utilization inside the VM with the actual CPU utilization as reported by the host system while the respective programs were running. A detailed setup of the experiments can be found in the appendix.

To cover a broad range of virtualization techniques, we chose the two popular open source hypervisors XEN [23] and KVM [83]. Traditionally, XEN is a prominent representative of paravirtualization, which requires a modified guest operating system and privileged instructions to be replaced by so-called hypercalls. Moreover, XEN is the primary virtualization technique used by Amazon EC2 [102]. In contrast to that, KVM has a large

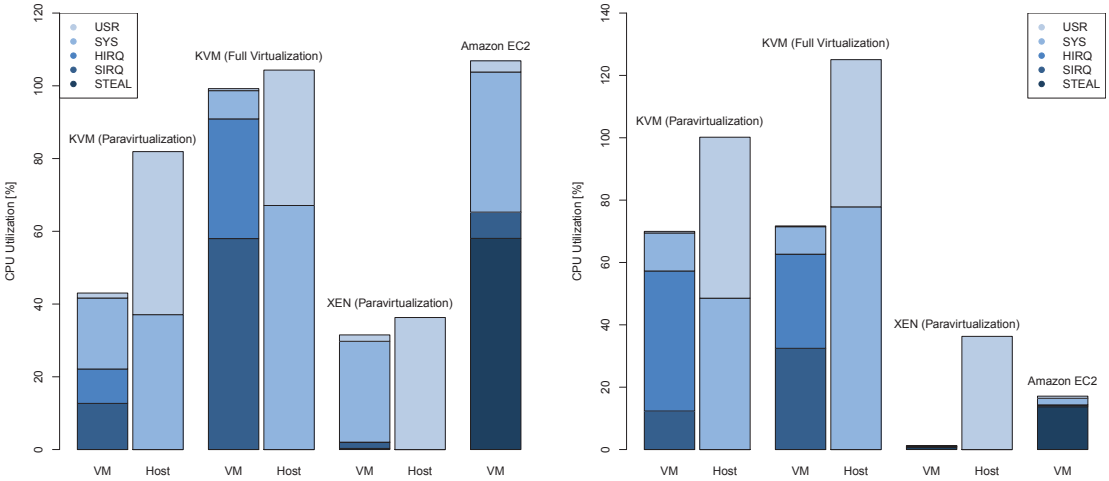
installed base with the software component QEMU [28] being used for the actual device emulation. Since QEMU does not require modifications to the guest operating system, KVM can be considered a representative of full virtualization with respect to I/O operations. However, we also evaluated the CPU overhead and accuracy of the displayed CPU utilization when running KVM with paravirtualization, using the modified virtio device drivers [113] instead of QEMU. As a baseline, we also examined the CPU utilization as displayed by VMs on Amazon EC2. For these experiments, however, we were unable to observe the CPU utilization as reported by the host system.

In order to monitor the CPU utilization inside the VMs we continuously queried the Linux system interface `/proc/stat` at an interval of one second. On the host system our monitoring scheme was dependent on the virtualization layer we used for the respective experiment. For KVM-based experiments we first determined the process ID of the corresponding QEMU process, afterwards traced the CPU utilization for that process using the `/proc/<process ID>/stat` interface, again at a sample interval of one second. For XEN-based experiments we used the management tool `xentop` to observe the CPU utilization that was accounted to the monitored `domU` from the perspective of the `dom0`.

Figure 2.2 illustrates the results of our experiments. Each of the four plots shows the average CPU utilization during one particular type of I/O operation (network send and receive, disk write and read) as reported by the operating system of the VM and the host system. The average has been calculated from at least 120 individual samples and is split into the fraction of time the CPU spent processing into user (USR) or kernel mode (SYS), serving hardware (HIRQ) or software interrupts (SIRQ). In case of XEN-based virtualization, STEAL denotes the amount of CPU time that the hypervisor has allocated to tasks other than the observed VM.

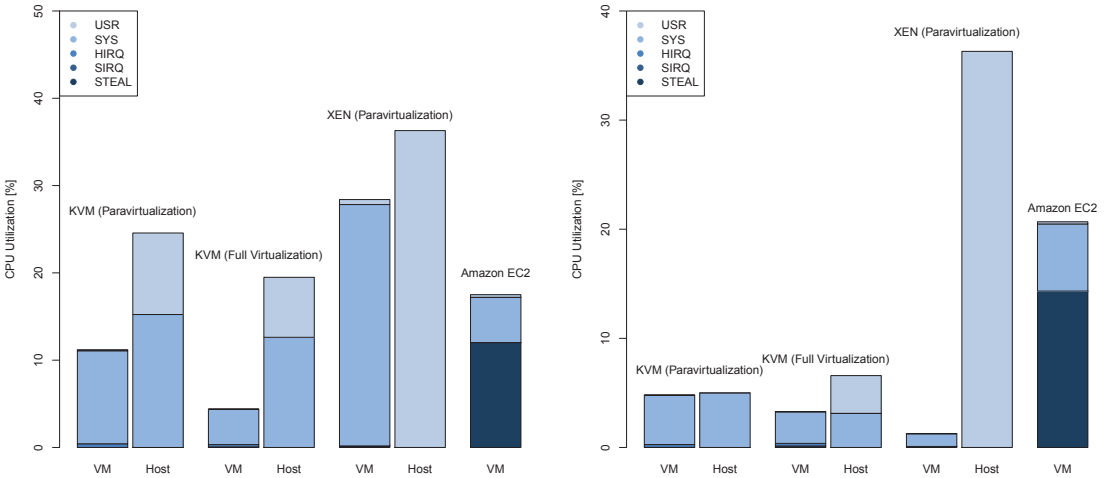
During all the experiments involving network transfer we used a TCP connection and made sure that the opposite part of the connection was an unvirtualized machine which was at least as fast as the observed VM. Hence, any potential performance bottleneck during these experiments was either induced by the network or the VM itself. For all experiments including disk I/O we used raw I/O API to avoid caching effects inside the VM as far as possible.

In sum, our experiments revealed large variations in the CPU overhead caused by I/O operations across the different virtualization techniques. In particular, we found network I/O under KVM (both using paravirtualization and full virtualization) to have a significant CPU overhead which is reflected by the CPU spending considerable amounts of time handling software (SIRQ) and hardware interrupts (HIRQ). In fact, in case of full virtualization, the achievable network throughput was limited by the CPU performance. In addition, the display of the CPU utilization inside the VM is often significantly lower than the actual CPU utilization reported by the host system. This discrepancy is not specific to a particular type of I/O operation or virtualization technique. It can be found



(a) Network I/O (send operation)

(b) Network I/O (receive operation)



(c) Disk I/O (write operation)

(d) Disk I/O (read operation)

Figure 2.2.: CPU overhead for executing I/O operations inside different VMs and accuracy of displayed CPU utilization during those operations.

across all considered I/O operations and virtualization techniques. While for some I/O operations the discrepancy in the reported CPU utilization is rather small (e.g. network send operation using KVM (full virtualization) or XEN), for others (e.g. network send operation using KVM (paravirtualization) or disk read operation using XEN) the gap can grow up to a factor of 15.

Besides the general impact of virtualization on the CPU performance, the CPU performance of VMs running on Amazon EC2 has recently caught the attention of several researchers. Walker [135] as well as Ostermann et al. [102] evaluated Amazon's platform with regard to scientific computing. Ostermann et al. describe the overall CPU performance on Amazon EC2 to be mixed, with excellent addition but poor multiplication capabilities. Both publications, the one from Walker and from Ostermann et al. conclude that MPI-based applications can run significantly slower on Amazon EC2 machines compared to native cluster setups with comparable hardware characteristics. However, since MPI-based applications also depend on low-latency network interconnects, those comparisons only provide little information about the pure CPU performance.

Wang and Ng [136] analyzed Amazon EC2's scheduling strategies for VMs. Through a set of microbenchmarks the authors discovered that the small, inexpensive VM types typically only receive a 40-50% share of the physical processor. In practice, this results in frequent interruptions of such VMs in the order of tens of milliseconds. These interruptions, in turn, also negatively affect the machines' network performance.

Schad et al. [114] studied the CPU performance on Amazon EC2 with special respect to performance variations. Their analysis revealed that VMs of the same type (with the same advertised hardware characteristics) may be hosted on different generations of host systems. This can lead to significant performance discrepancies between two different VMs of the same type. Since the customer in general cannot influence the target host system for his VMs, the authors conclude that performance predictability on Amazon EC2 is currently hard to achieve.

2.3.2. I/O Performance Characteristics

The I/O performance characteristics of IaaS clouds or virtualized environments in general can be considered on three different levels of scope. The first level of scope is limited to a single VM. As pointed out in the previous subsection, within this scope, the virtualization layer itself can have an impact on the I/O performance, depending on the CPU overhead that occurs for the concrete virtualization approach. The second level of scope involves two or more colocated VMs, i.e., VMs which are hosted on the same physical server. Finally, the third level of scope covers large sets of VMs which are potentially hosted on many different servers and even racks within a cloud data center.

Since all three levels of scope are important with respect to efficient parallel data processing on IaaS platforms, the thesis will now discuss them one after the other.

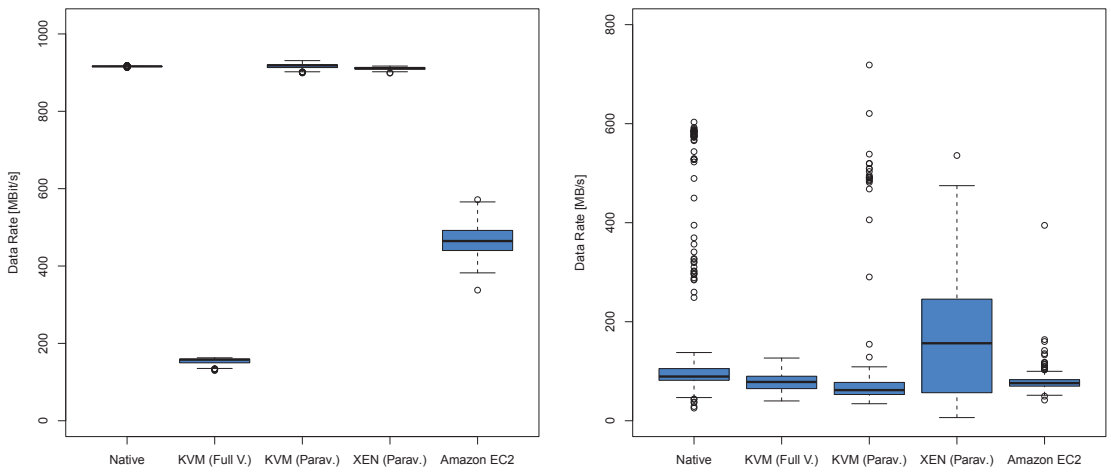
Level 1: Single VM Performance Characteristics

As already mentioned in the previous subsection, even if only a single VM is executed on a physical server, its I/O performance may be degraded as a result of the additional management and translation work the CPU has to perform. The exact performance penalty thereby depends on the concrete virtualization technique and I/O operation.

In order to compare the I/O performance of various virtualization approaches to the one of a native, unvirtualized system, we conducted several microbenchmarks on our local IaaS clouds testbed [69]. Similar to the CPU microbenchmarks from the previous subsection, VMs running on KVM (full virtualization and paravirtualization), XEN (paravirtualization) as well as VMs running externally on Amazon EC2 were contrasted. We modified our set of auxiliary programs to records timestamps after every 20 MB of generated or consumed I/O data, respectively. With the help of these timestamps we then calculated the I/O data rate as it appeared from within the VM. In total, each auxiliary program either produced or consumed 50 GB of data. Like in the previous experiments, the underlying physical host was fully dedicated to the observed VM. A detailed description on the testbed is again included in the appendix.

Figure 2.3 illustrates the results of the microbenchmarks for network and disk I/O. In terms of network throughput, the machines running with paravirtualization (Figure 2.3a) on our local cloud testbed achieved an I/O performance comparable to the native system. However, for the KVM-based VM with unmodified device drivers (KVM full virtualization), we observed a significant drop in network throughput to about 160 MBit/s. The degradation can be explained by the high amount of CPU overhead that occurs for this kind of I/O virtualization. It also complies with the CPU bottleneck described for this case in the previous subsection. The network throughput for the experiments we conducted on Amazon EC2 averaged to approximately 500 MBit/s. Although this data rate is significantly lower than the one we observed for XEN-based VMs on our local cloud testbed, the results correspond to previous evaluations for the same type of VM [136].

Besides the mean I/O throughput, Figure 2.3 also shows the distribution of the data rates measured within the individual VMs and on the native baseline system. For the network I/O experiments, the throughput variances introduced by the different virtualization layers on our local IaaS testbed are only marginal. Only the network experiments on Amazon EC2 showed throughput fluctuations in the range of approximately 200 MBit/s. They are likely to be caused by Amazon's VM scheduling strategies [136]. However, when writing data to the VM's disk (Figure 2.3b), the distribution reveals another important



(a) I/O throughput (network)

(b) I/O throughput (disk write)

Figure 2.3.: I/O throughput and variance as observed within different types of VMs.

aspect of I/O performance in virtualized environments that must be carefully considered, namely *caching*.

The disk I/O measurements conducted on KVM-based VMs (both full and paravirtualization) as well as on Amazon EC2 showed a throughput fluctuation which is comparable to the one of our native baseline system. However, with the XEN-based VM, which we instantiated on our local cloud testbed, we witnessed significant caching effects. Due to these caching effects the data rate inside the VM occasionally appeared to be exceedingly high. In fact, the data was only buffered inside the host system's main memory. Periodically, when the host system decided to actually flush the buffered data to disk, the data rate displayed inside the VM dropped to only a few MB/s. As a result of these caching effects, the average data throughput for the XEN-based disk I/O experiments also spuriously appears to be higher compared to the experiments on the native or KVM-based systems in the plot. However, after the 50 GB of data had been written to the VM's disk, large portions of the data had not actually been transferred to the physical hard disk, but still remained inside the host system's main memory.

Similar caching effects have also been reported for VMs on Amazon EC2. Through a series of microbenchmarks, Ostermann et al. determined the capacity of the memory-based disk cache for all types of VMs to be about four to five GB [102]. However, our own benchmarks on Amazon EC2 do not fully support these numbers. In the course of

our experiments involving an EC2 VM of type "m1.small", we only witnessed negligible caching effects (cf. Figure 2.3). Possible explanations for this discrepancy might be recent updates to Amazon's software stack.

Finally, the performance characteristics of accessing Amazon's storage services from VMs hosted on Amazon EC2 were examined by Palankar et al. [103] and Shafer [116], respectively. Palankar et al. measured the data access performance for reads from VMs on Amazon EC2 to the S3 storage service. According to the authors' results, the read performance of Amazon S3 suffers from the transaction overhead for data objects less than one MB in size. However, for larger objects, the authors reported throughput rates of approximately 20 to 30 MB/s per object and EC2 instance, depending on the experiment. Shafer examined the performance of Amazon EBS. In his paper, the author reports data transfer rates of roughly 43 MB/s for write and 68 MB/s for read operations.

Level 2: Performance Characteristics of Colocated VMs

In order to run their data centers in an economical manner, operators of IaaS clouds typically accommodate multiple VMs on one physical server. Those VMs, which share the same physical server, are also called colocated VMs.

As a result of the colocation, the I/O performance characteristics of the VMs are no longer independent of each other. For example, the I/O performance of one VM which requires to read large portions of data from the underlying server's hard disk may be detrimentally affected by a colocated VM which causes the hard disk to do frequent random disk accesses.

Improving the fairness of colocated VMs has been an area of vivid research in recent years. By default, XEN used the so-called Borrowed Virtal Time (BVT) algorithm [48] to schedule the execution of multiple VMs. The algorithm is an adaptation of a traditional operating system scheduler and aims at providing low-latency for real-time and interactive applications, yet weighted sharing of the CPU across applications. However, as pointed out by Kesavan et al. [82], the problem of scheduling VMs is more complex, because, in contrast to an application, the guest operating system inside a VM typically incorporates its own proprietary internal resource management. For example, interrupting a VM without regard to its network behavior can cause considerable jitter in the round-trip time (RTT) measurements carried out by the guest operating system's TCP subsystem [82]. This jitter may lead to unnecessary triggering of TCP's congestion avoidance mechanisms which, in turn, lowers the available network throughput.

As a remedy to this problem, several approaches to include the I/O characteristics of VMs into the scheduling decision have been proposed for the popular XEN hypervisor [82,

62, 79, 146, 67]. However, at this point in time, none of the proposed enhancements has been officially included in the XEN source code [40]. If at all, they are available as inofficial patches which must be applied to the XEN source code by each XEN user individually. Since the maturity of these patches is generally unknown, it is highly doubtful whether they will be used in a commercial setting like an IaaS cloud. As a result, I/O sharing can currently result in significant and unpredictable levels of degradation in VM performance [82].

These significant and unpredictable levels of performance degradation have also been confirmed for particular types of VMs running on Amazon EC2 [136]. Although Amazon EC2 does not publish any details on their data center setup, Wang et al. found through various experiments that the processor sharing which Amazon EC2 uses for the inexpensive, small VM types causes very unstable TCP and UDP throughput. The authors observed large throughput variations for these types of VMs. According to their paper, the TCP/UDP throughput experienced by applications can fluctuate between one GB/s and zero, even at a timescale of tens of milliseconds. Moreover, the authors describe abnormally large packet delay variations which can be hundred times larger than the propagation delay between two end hosts [136].

The disk I/O performance on Amazon EC2 was evaluated by Schad et al. [114] and Ostermann et al. [102]. According to the results of Ostermann et al., all types of VMs in general offer better performance for sequential operations compared to similar, unvirtualized commodity systems. Schad et al. also analyzed the variance of the disk performance. They found that disk performance can vary significantly across different instantiations of the same VM type. The authors assume different types of hard disks inside Amazon data centers to be responsible for this effect.

Level 3: Performance Characteristics across Different Servers

As explained in the introduction, current platforms for massively parallel data processing build upon an architectural paradigm which favors a large set of inexpensive compute nodes over expensive servers. The communication among these compute nodes must be carried out over the network. Consequently, the network bandwidth becomes a scarce resource in such setups [43].

In order to unburden the network and avoid possible network bottlenecks, a variety of parallel data processing frameworks try to exploit data locality, i.e., to restrict data transfers to specific parts of the network as far as possible. For example, Apache Hadoop [124], the open source implementation of Google's MapReduce framework [43], attempts to read remote data blocks from those nodes which are in close proximity to the one requiring the data. The proximity between the compute nodes is thereby typically determined

with respect to the physical network topology. It is considered to be the number of internal network components a data packet must traverse from one node to the other.

While this type of proximity is easy to determine in a static cluster, it imposes some serious obstacles for today's IaaS clouds. In an IaaS cloud, the underlying network topology, i.e., the way the individual VMs are physically interconnected with each other, is usually not exposed to the customer. As a result, the proximity between two VMs cannot be expressed easily. Although diagnosis tools like `traceroute` can potentially be used to sketch a course-grained network topology connecting the customer's individual VMs, these tools rely on the cooperation of the internal network nodes [148]. Moreover, they fail to identify link-layer network components like switches or bridges which also play an important role for exploiting data locality.

Especially knowledge about network bridges which connect colocated VMs to the physical network can be highly valuable with regard to parallel data processing. In a small microbenchmark using the well-known network measurement tool `Iperf` [128], I found data transfers between two colocated KVM-based VMs to achieve throughput rates of up to 1.5 GBit/s. In contrast to that, the transfer rates between two VMs which were hosted on different physical servers summed up to at most 950 MBit/s, because, unlike in the first case, data actually had to be sent across the real Ethernet network.

Although there exist different protocols to query link layer network topologies (such as LLDP [72]), these protocols primarily aim at network diagnostics and can potentially reveal security-relevant information about the cloud operator's IT infrastructure. I am not aware of a single cloud operator who offers support for such protocols. Moreover, even if those services were enabled, there may be legal restrictions like Amazon's terms of service [15] under which such probes could be interpreted as unauthorized network access [111].

Another important aspect of data locality in IaaS clouds is the fact that the location of a VM can potentially change throughout its lifetime due to migration [97]. A cloud operator may migrate a VM from one server to another for several reasons, for example to consolidate the number of running servers or to guard against an impending hardware outage. With regard to parallel data processing on top of IaaS clouds this means that the transfer cost for a pair of VMs can potentially change during a processing job.

2.4. Summary

In this chapter the thesis highlighted the characteristics of IaaS clouds, ranging from economic aspects to technical properties. With respect to the mission statement of this thesis, the chapter helped to draw a clearer picture of the opportunities for efficient

parallel data processing in IaaS clouds but also of the technical challenges that must be taken into account.

In terms of the opportunities, namely improving the efficiency of a processing job by dynamically adapting the number and the type of worker nodes at runtime, the chapter underlined that essentially all IaaS clouds today fulfill the necessary economical and technical requirements to support the use cases motivated in the introduction:

First, all IaaS platforms I considered featured a clearly defined set of VM types, each type with distinct hardware characteristics and a distinct price per hour. The offerings of the respective cloud operators typically have a validity period of at least several months, if not years, so that the hardware and cost properties of the individual VM types could be fed into a parallel data processing framework and used as a basis for resource management and scheduling decisions. For offerings which are subject to frequent price fluctuations, such as Amazon's spot instances, the respective cloud operators usually provide an API to dynamically retrieve the latest pricing information.

Second, all considered IaaS clouds provide well-documented APIs to enable external applications to manage the set of allocated computing resources on behalf of the customer. Several measurements on Amazon EC2 have confirmed that the time span from requesting a new VM until it becomes available to the customer is normally below three minutes. Both aspects are important prerequisites to quickly respond to changes in the workload of a parallel data processing job.

Finally, most IaaS clouds offer dedicated storage services with a separate pricing model besides their compute service. For data-intensive processing jobs this means that a customer can transfer the job's input data to the cloud provider or store the job's output data without the need to run the potentially more expensive computing resources.

However, despite the good starting point for exploiting dynamic resource allocation, the discussion on the performance characteristics of a IaaS cloud also underlined the initial concerns for efficient parallel data processing on top of the new platform.

In particular, the chapter analyzed how the hardware virtualization, which is used to facilitate the cloud's rapid resource provisioning, can lead to significant and unpredictable levels of degradation in the performance of individual VMs. With respect to parallel data processing, this can easily lead to bottlenecks in the processing chain which may leave large parts of the allocated computing resources underutilized and outweigh the benefits of the dynamic resource allocation. Moreover, the chapter illustrated how the overhead of I/O virtualization or caching effects can lead to spurious displays of the system performance inside a VM. This is especially a problem for all mechanisms that have to trade off CPU against I/O performance, such as adaptive compression schemes.

For the remainder of this thesis, I can therefore conclude that, besides their ability of rapid resource provisioning, common IaaS platforms are characterized by a considerable loss of control over the physical hardware compared to classic cluster environments. Although there exist different approaches to improve a VM's control over the underlying hardware (especially new hardware features like Intel VT-d [73]), it is currently unclear to what extent commercial cloud providers will adopt these technologies. Recently, Amazon EC2 announced two new VM types for high performance computing which leverage these new hardware features [18]. However, there has not been any comprehensive analysis on the performance benefits yet.

As a result, a data processing framework designed for today's IaaS clouds should rather embrace the possible performance fluctuations which may arise from the loss of hardware control and attempt to mitigate them on a software level.

3. Exploiting Dynamic Resource Allocation

Contents

3.1. Design Principles **28**
3.2. The Nephele Parallel Data Processing Framework **29**
3.2.1. Architecture 29
3.2.2. Job Description 31
3.2.3. Job Scheduling and Execution 33
3.3. Parallelization and Scheduling Strategies **36**
3.3.1. Finding Suitable Degrees of Parallelism and VM Types 36
3.3.2. Automatic VM Allocation and Deallocation 38
3.4. Evaluation **39**
3.4.1. Experiment 1: MapReduce and Hadoop 40
3.4.2. Experiment 2: MapReduce and Nephele 41
3.4.3. Experiment 3: DAG and Nephele 44
3.4.4. Results 46
3.5. Related Work **50**
3.6. Summary **52**

As motivated in the introduction, the ability of IaaS clouds to provide large sets of possibly heterogeneous compute nodes in a matter of seconds enable new use cases for parallel data processing which have not been possible in classic cluster setups before. Moreover, the pay-as-you-go pricing model of clouds adds a very interesting notion of monetary cost to this field of large-scale distributed applications and directly incentivizes customers to use the rented resources in an economical fashion.

The current state of the art of parallel data processing on top of IaaS platforms, however, is rather an imitation of the static, homogeneous compute cluster era. For example, the current work flow to execute a parallel data processing job with Amazon Elastic MapReduce [14], probably the best known offering for cloud-based data processing at the moment, requires the cloud customer to choose a fixed number of VMs of a particular type before the start of the job's execution. Although the processing job might be composed of several invididual subjobs, the number or type of the involved VMs cannot be changed once the main job has been launched. The main reason for this is that Hadoop,

the technical foundation of the service, does not support these kinds of operations. Thus, all opportunities for improving the efficiency of parallel data processing, either in terms of processing time or cost, by means of the cloud's new abilities remain wasted.

Following the discussion on the characteristics of IaaS clouds, the thesis will therefore now elaborate on design principles a framework for parallel data processing has to meet in order to be able to exploit the cloud's dynamic resource allocation in the course of a processing job. Based on these design principles, I then present our new framework for parallel data processing called *Nephele* [138, 139].

Nephele is the first data processing framework to explicitly exploit the dynamic resource allocation and resource heterogeneity offered by today's IaaS platforms for both task scheduling and execution. Particular tasks of a processing job can be assigned to different types of VMs which are automatically instantiated and terminated during the job execution. This chapter will introduce Nephele's basic architecture and programming abstraction. In order to illustrate the benefits of the dynamic resource allocation, we have performed several evaluations of MapReduce-inspired processing jobs on an IaaS cloud system and compared the results to the popular data processing framework Hadoop.

3.1. Design Principles

Although the design of frameworks for massively parallel data processing may differ in various ways, there are three fundamental design principles such a framework has to fulfill to be able to take advantage of the cloud's rapid resource provisioning and resource heterogeneity:

- First, the framework's scheduler (or in general the respective framework's component which is responsible for resource management) must become aware of the cloud environment a job should be executed in. It must know about the different types of available VMs as well as their cost and be able to allocate or destroy them on behalf of the cloud customer. Technically, this entails that the scheduler must know the network address of the Cloud Controller as introduced in Section 2.2. Moreover, the scheduler must implement the cloud provider's API to be able to adapt the set of allocated computing resources in the course of the job execution.
- Second, the paradigm used to describe jobs for the respective data processing framework must be powerful enough to express dependencies between the different tasks the job consists of. The system must be aware of which task's output is required as another task's input and on what node the respective data currently resides. Otherwise the framework's scheduler cannot decide at what point in time a particular VM is no longer needed and deallocate it. The MapReduce pattern

implemented by frameworks like MapReduce [43] and Hadoop [124] is a good example of a problematic paradigm here because data dependencies beyond a single MapReduce job cannot be expressed. As a result, it is in general not possible to reduce the number of nodes between two consecutive jobs, since there is always the risk that the deallocated nodes contained important intermediate results which might have been required by the next MapReduce job.

- Finally, the scheduler of a processing framework must be able to determine which task of a job should be executed on which type of VM and, possibly, how many of those. This information could be either provided externally, for example as an annotation to the job description, or deduced internally, for instance from collected statistics, similarly to the way database systems try to optimize their execution plan over time [120].

While the three aspects discussed above are important cornerstones towards exploiting dynamic resource allocation in the scope of parallel data processing, they leave different degrees of freedom in the concrete design of a processing framework. In the next section I will therefore present a concrete data processing framework built upon these principles and highlight in which ways they have influenced particular design decisions.

3.2. The Nephele Parallel Data Processing Framework

In this section I will now present the parallel data processing framework Nephele [138, 139] as a concrete implementation of the design principles discussed in the previous section. The section will introduce Nephele's basic architecture, its abstraction for describing jobs as well as the most important internal scheduling data structures.

3.2.1. Architecture

In order to describe Nephele's basic architecture and its interaction with the underlying IaaS platform, I will build on the components introduced in Section 2.2.

As illustrated in Figure 3.1, Nephele's architecture follows a classic master-worker pattern. Before submitting a Nephele job, a cloud customer must start a VM which runs the so-called Job Manager (JM). The Job Manager receives jobs from the customer's job client, is responsible for scheduling them, and coordinates their execution. It is not required to run it on a node inside the IaaS cloud, but for simplicity I will assume this.

Following the design principles from the previous section, the Job Manager is the component in the data processing framework which is capable of communicating with the

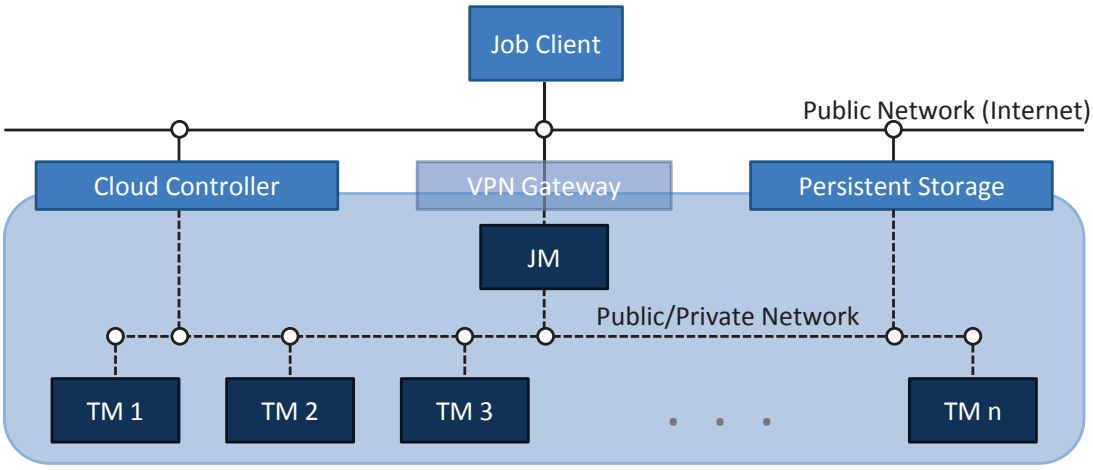


Figure 3.1.: Structural overview of Nephele running in an IaaS cloud.

Cloud Controller. Hence, it knows about the different types of VMs the underlying IaaS cloud offers as well as their respective price per hour. In addition, it implements the cloud's specific management API. By means of the Cloud Controller the Job Manager can therefore allocate or deallocate VMs according to the current job execution phase.

The actual execution of tasks which a Nephele job consists of is carried out by a set of VMs. Each VM runs a so-called Task Manager (TM). A Task Manager receives one or more tasks from the Job Manager at a time, executes them, and after that informs the Job Manager about their completion or possible errors. Unless a job is submitted to the Job Manager, the set of VMs (and hence the set of Task Managers) is expected to be empty. Upon job reception the Job Manager then decides, depending on the job's particular tasks, how many and what type of VMs the job should be executed on, and when the respective VMs must be allocated/deallocated to ensure an efficient processing. The current strategies for these decisions are highlighted in Section 3.3. The newly allocated VMs boot up with a previously compiled machine image. The image is configured to automatically start a Task Manager and register it with the Job Manager. Once all the necessary Task Managers have successfully contacted the Job Manager, it triggers the execution of the scheduled job.

Initially, the VM image used to boot up the Task Managers is blank and does not contain any of the data the Nephele job is supposed to operate on. Therefore, the input data of the respective job is expected to be stored on the cloud's persistent storage service and to be accessible to the individual VMs in similar ways as described in Section 2.1.

3.2.2. Job Description

Jobs in Nephele are expressed as a directed acyclic graph (DAG). Each vertex in the graph represents a task of the overall processing job, the graph's edges define the communication flow between these tasks. We decided to use DAGs to describe processing jobs for two major reasons:

The first reason is that DAGs allow tasks to have multiple input and multiple output edges. This tremendously simplifies the implementation of classic data combining functions like, for example, join operations [147]. The second reason reflects one of the design principles highlighted in the previous section: The DAG's edges explicitly model the communication paths of the processing job. As long as the particular tasks only exchange data through these designated communication edges, Nephele can always keep track of what VM might still require data from what other VMs and which VM can potentially be shut down and deallocated.

Defining a Nephele job comprises three mandatory steps: First, the cloud customer must write the user code for each task of his processing job or select it from an external library. Second, the user code must be assigned to a vertex. Finally, the vertices must be connected by edges to define the communication paths of the job.

Tasks are expected to contain sequential code and process so-called records, the primary data unit in Nephele. Developers can define arbitrary types of records. From a developer's perspective records enter and leave the task program through input or output gates. Those input and output gates can be considered endpoints of the DAG's edges which are defined in the following step. Regular tasks (i.e., tasks which are later assigned to inner vertices of the DAG) must have at least one or more input and output gates. Contrary to that, tasks which either represent the source or the sink of the data flow must not have input or output gates, respectively.

After having specified the code for the particular tasks of the job, the cloud customer must define the DAG to connect these tasks. This DAG is called the *Job Graph*. The Job Graph maps each task to a vertex and determines the communication paths between them. The number of a vertex's incoming and outgoing edges must thereby comply with the number of input and output gates defined inside the tasks. In addition to the task to execute, input and output vertices (i.e., vertices with either no incoming or outgoing edge) can be associated with a URL pointing to external storage services in order to read or write input or output data, respectively. Figure 3.2 illustrates a simple Job Graph. It only consists of one input, one task, and one output vertex.

One major design goal of Job Graphs has been simplicity: Cloud customers should be able to describe tasks and their relationships on an abstract level. Therefore, the Job Graph does not explicitly model task parallelization and the mapping of tasks to VMs.

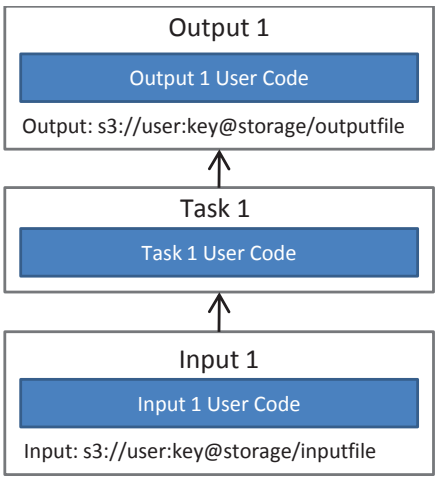


Figure 3.2.: An example of a Job Graph in Nephele.

However, cloud customers who wish to influence these aspects can provide annotations to their job description. These annotations include:

- **Number of subtasks:** A developer can declare his task to be suitable for parallelization. Cloud customers that include such tasks in their Job Graph can specify how many parallel subtasks Nephele should split the respective task into at runtime. Subtasks execute the same task code, however, they typically process different fragments of the data.

- **Number of subtasks per VM:** By default each subtask is assigned to a separate VM. In case several subtasks are supposed to share the same VM, the cloud customer can provide a corresponding annotation with the respective task.

- **Sharing VMs between tasks:** Subtasks of different tasks are usually assigned to different (sets of) VMs unless prevented by another scheduling restriction. If a set of VMs should be shared between different tasks, the cloud customer can attach a corresponding annotation to the Job Graph.

- **Channel types:** For each edge connecting two vertices the cloud customer can determine a channel type. Before executing a job, Nephele requires all edges of the original Job Graph to be replaced by at least one channel of a specific type. The channel type dictates how records are transported from one subtask to another at runtime. Currently, Nephele supports network, file, and in-memory channels. The choice of the channel type can have several implications on the entire job schedule. A more detailed discussion on this is provided in the next subsection.

- **VM type:** A subtask can be executed on different VM types which may be more or less suitable for the considered program. Therefore, we have developed special annotations task developers can use to characterize the hardware requirements of their code. However, a cloud customer who simply utilizes these annotated tasks can also overwrite the developer's suggestion and explicitly specify the VM type for a task in the Job Graph.

If the cloud customer omits to augment the Job Graph with these specifications, Nephele's scheduler applies default strategies which are discussed later on in Section 3.3. Once the Job Graph is specified, the cloud customer submits it to the Job Manager, together with the credentials he has obtained from his cloud operator. The credentials are required since the Job Manager must allocate/deallocate VMs during the job execution on behalf of the cloud customer.

3.2.3. Job Scheduling and Execution

After having received a valid Job Graph from the cloud customer, Nephele's Job Manager transforms it into a so-called *Execution Graph*. An Execution Graph is Nephele's primary data structure for scheduling and monitoring the execution of a Nephele job. Unlike the abstract Job Graph, the Execution Graph contains all the concrete information required to schedule and execute the received job on the IaaS platform. It explicitly models task parallelization and the mapping of tasks to VMs. Depending on the level of annotations the user has provided with his Job Graph, Nephele may have different degrees of freedom in constructing the Execution Graph. Figure 3.3 shows one possible Execution Graph constructed from the previously depicted Job Graph (Figure 3.2). Task 1 is, for example, split into two parallel subtasks which are both connected to the task Output 1 via file channels and are all scheduled to run on the same VM. The exact structure of the Execution Graph is explained in the following:

In contrast to the Job Graph, an Execution Graph is no longer a pure DAG. Instead, its structure resembles a graph with two different levels of details, an abstract and a concrete level. While the abstract graph describes the job execution on a task level (without parallelization) and the scheduling of VM allocation/deallocation, the concrete, more fine-grained graph defines the mapping of subtasks to VMs and the communication channels between them. On the abstract level, the Execution Graph equals the cloud customer's Job Graph. For every vertex of the original Job Graph there exists a so-called *Group Vertex* in the Execution Graph. As a result, Group Vertices also represent distinct tasks of the overall job, however, they cannot be regarded as executable units. They are used as a management abstraction to control the set of subtasks the respective task program is split into. The edges between Group Vertices are only modeled implicitly as

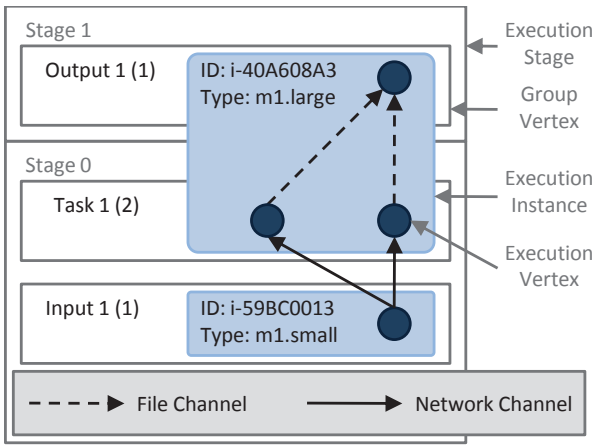


Figure 3.3.: An Execution Graph created from the original Job Graph.

they do not represent any physical communication paths during the job processing. For the sake of presentation, they are also omitted in Figure 3.3.

As discussed in Section 2.1, IaaS clouds usually do not guarantee the availability of computing resources at all times. Since Nephele attempts to improve a job's cost efficiency by allocating the required VMs as late as possible in the course of its execution, this situation may be problematic. For example, a job might require to allocate a VM of a particular type in the middle of its execution. If no such VM is available at that moment, the job is unable to proceed. Moreover, the already allocated machines cannot be shut down because the user code which runs inside the tasks on these nodes may be in an inconsistent state and important intermediate results would be lost.

In order to cope with this problem, Nephele separates the Execution Graph into one or more so-called *Execution Stages*. An Execution Stage must contain at least one Group Vertex. Its processing can only start when all the subtasks included in the preceding stages have been successfully processed. Based on this, Nephele's scheduler ensures the following three properties for the entire job execution: First, when the processing of a stage begins, all VMs required within the stage are allocated. Second, all subtasks included in this stage are set up (i.e., sent to the corresponding Task Managers along with their required libraries) and ready to receive records. Third, before the processing of a new stage, all intermediate results of its preceding stages are stored in a persistent manner. Hence, Execution Stages can be compared to checkpoints. In case a sufficient number of resources cannot be allocated for the next stage, they allow a running job to be interrupted and later on restored when enough spare resources have become available.

The concrete level of the Execution Graph refines the job schedule to include subtasks and their communication channels. In Nephele, every task is transformed into either

exactly one, or, if the task is suitable for parallel execution, at least one subtask. In order to complete a task successfully, each of its subtasks must be successfully processed by a Task Manager. Subtasks are represented by so-called *Execution Vertices* in the Execution Graph. They can be considered the most fine-grained executable job unit. In order to simplify management, each Execution Vertex is always controlled by its corresponding Group Vertex.

Nephele allows each task to be executed on its own type of VM, so the characteristics of the requested VMs can be adapted to the demands of the current processing phase. In order to reflect this relation in the Execution Graph, each subtask must be mapped to a so-called *Execution Instance*. An Execution Instance is defined by an ID and an instance type representing the hardware characteristics of the corresponding VM. It is a scheduling stub that determines which subtasks have to run on what VM (type). As described in Section 3.1, a list of available VM types together with their cost per time unit is expected to be accessible for Nephele's scheduler. Moreover, Nephele assumes that an individual instance type can be referenced by a single identifier string like "m1.small", similar to the way Amazon EC2 deals with its different VM types.

Before processing a new Execution Stage, the scheduler collects all Execution Instances from that stage and tries to replace them with matching VMs from the IaaS cloud. If all required VMs could be allocated, the subtasks are distributed among them and set up for execution.

On the concrete level, the Execution Graph inherits the edges from the abstract level, i.e., edges between Group Vertices are translated into edges between Execution Vertices. In case of task parallelization, when a Group Vertex contains more than one Execution Vertex, the developer of the consuming task can implement an interface which determines how to connect the two different groups of subtasks. The actual number of channels that are connected to a subtask at runtime is hidden behind the task's respective input and output gates. However, the user code can determine the number if necessary.

Nephele requires all edges of an Execution Graph to be replaced by a communication channel before processing can begin. The type of the channel determines how records are transported from one subtask to another. Currently, Nephele features three different types of channels, all of them put different constrains on the Execution Graph.

- **Network channels:** A network channel lets two subtasks exchange data via a TCP connection. Network channels allow pipelined processing, so the records emitted by the producing subtask are immediately transported to the consuming subtask. As a result, two subtasks connected via a network channel may be executed on different VMs. However, since they must be executed at the same time, they are required to run in the same Execution Stage.

- **In-memory channels:** Similar to a network channel, an in-memory channel also enables pipelined processing. However, instead of using a TCP connection, the respective subtasks exchange data using the VM's main memory. An in-memory channel typically represents the fastest way to transport records in Nephele, however, it also implies most scheduling restrictions: The two connected subtasks must be scheduled to run on the same VM and run in the same Execution Stage.
- **File channels:** A file channel allows two subtasks to exchange records via the local file system. The records of the producing task are first entirely written to an intermediate file and afterward read into the consuming subtask. Nephele requires two such subtasks to be assigned to the same VM. Moreover, the consuming Group Vertex must be scheduled to run in a higher Execution Stage than the producing Group Vertex. In general, Nephele only allows subtasks to exchange records across different stages via file channels because this channel type is the only one which stores the intermediate records in a persistent manner.

3.3. Parallelization and Scheduling Strategies

As explained in the previous section, the Execution Graph is Nephele's primary data structure for scheduling and monitoring the execution of a job. Depending on the degree of user annotations, constructing an Execution Graph from the submitted Job Graph may leave different degrees of freedom to Nephele. Using this freedom to construct the most efficient Execution Graph (in terms of processing time or monetary cost) entails several research challenges. This section discusses these challenges as well as Nephele's current strategies to address them.

3.3.1. Finding Suitable Degrees of Parallelism and VM Types

According to the cost model of an IaaS cloud, renting a thousand CPU cores for an hour is no longer more expensive than renting a single CPU core for a thousand hours. As a result, it might be tempting for cloud customers to strive for shorter completion times of their jobs by increasing their level of parallelization. Of course, the vast majority of compute jobs cannot be parallelized indefinitely. At some level of parallelization, the I/O subsystem of the rented cloud resources (i.e., the bandwidth of the hard disk and the network links) will become an insuperable bottleneck.

Parallelization beyond that point will leave the rented CPU cores underutilized and unnecessarily increase the cost for executing the processing job on the one hand. On the other hand, too low degrees of parallelization might cause CPU bottlenecks in the

processing chain, which may slow down the execution of successive tasks and also leave large parts of the rented computing resources underutilized.

Dealing with this problem is difficult because each Nephele vertex is expected to contain sequential but arbitrary user code. As a result, Nephele initially cannot make any assumptions about the behavior of a task, neither about its computational complexity nor its I/O characteristics.

As a remedy, Nephele currently pursues the following strategy: Unless the user provides any job annotation which contains more specific instructions, each task of a Nephele job is executed with a parallelization level of one at its first run. Moreover, each task is by default assigned to a separate VM and connected via network channels to its predecessor and successor tasks. The default VM type to be used is the one with the lowest price per hour available in the IaaS cloud. However, during the job's execution, Nephele offers to monitor the job's individual tasks and thereby to learn about their CPU and I/O characteristics. Based on the gathered data, Nephele is also able to compute bottlenecks which have occurred in the course of the processing. Details on the job monitoring and the bottleneck detection algorithms are discussed in the next chapter.

Figure 3.4 illustrates a special graphical job viewer we devised to provide a cloud customer with immediate visual feedback on the CPU and I/O characteristics of his job and possible processing bottlenecks. In the current version of Nephele, a cloud customer can utilize this feedback to successively improve the scale-out and VM assignment of a job across different runs. For example, computational bottlenecks suggest a higher degree of parallelization for the affected tasks. In contrast to that, I/O bottlenecks provide hints to switch to faster channel types (like in-memory channels) which might, in turn, suggest employing larger VM types (i.e., VMs with more CPU cores).

However, since Nephele calculates a cryptographic signature for each task, it is also conceivable to identify recurring tasks by means of this signature and to use the previously recorded feedback data as one aspect to automatically derive reasonable degrees of parallelism for them. Technically, it would also be possible to respond to any detected CPU or I/O bottleneck immediately within the course of a job execution. For example, Nephele could respond to CPU bottlenecks by successively allocating new VM from the IaaS cloud and launching new subtasks of the respective task until the bottleneck situation is resolved. Nevertheless, it is important to recall that the user code within a task is typically arbitrary. Hence, the behavior of a task can change during its execution time so that newly created subtasks may soon become idle again. Moreover, general user code cannot be expected to cope with arbitrarily creating and removing instances of it. Special user annotations can possibly help to resolve both of these issues. Further ideas on these annotations are discussed in Chapter 7.

Figure 3.4.: Nephele's job viewer provides immediate visual feedback on task characteristics and possible processing bottlenecks.

3.3.2. Automatic VM Allocation and Deallocation

As indicated in the previous section, the points in time when Nephele allocates new VMs for a processing job or releases machines which are no longer needed are on principle determined by the Execution Stages a job is separated into. However, there are several refinements to Nephele's allocation strategy that shall be discussed in this subsection.

In general, new VMs for a processing job are always allocated at the beginning of a new Execution Stage. This guarantees that all intermediate results have been stored in a persistent manner, so in case the IaaS cloud is temporarily unable to fulfill Nephele's resource request, the job execution can be interrupted in a consistent state and possibly

restored later on.

However, depending on the job's scale-out as well as the concrete IaaS cloud Nephele uses to retrieve VMs from, the risk of experiencing resource unavailability may be negligibly small. For these situations Nephele also features a so-called *lazy initialization mode*. In this lazy initialization mode, the individual subtasks of a task are not scheduled until their preceding subtasks actually attempt to send them any records. Consequently, the allocation of the corresponding VMs which run these subtasks can also be postponed.

The lazy initialization mode is particularly beneficial for jobs which include one or more dams, i.e., user code which prevents the records from flowing through the graph like in a pipeline. For example, a sort task typically represents a dam as it requires to fully consume its fraction of the input data before it can output any records. Depending on the number and concrete nature of these dams in a processing job, Nephele's lazy initialization mode may therefore lead to later allocation times for VMs of particular tasks and further contribute to the job's cost efficiency.

While the allocation time of VMs is ultimately determined by the latest possible start times of the assigned subtasks, there are different possible strategies for VM deallocation. In order to reflect the fact that most IaaS platforms charge their customers for VM usage by the hour, Nephele integrates the possibility to reuse VMs. To do so, it keeps track of the VMs' allocation times. A machine of a particular type which no longer runs any subtask is not immediately deallocated if a VM of the same type is required either later in the same Execution Stage or an upcoming one. Instead, Nephele keeps the VM allocated until the end of its current lease period. If the machine is again required before the end of that period, it is reassigned to a new Execution Vertex, otherwise it is deallocated early enough not to cause any additional cost.

3.4. Evaluation

In this section, the thesis presents first performance results of Nephele and compares them to the data processing framework Hadoop. We have chosen Hadoop as our competitor because it is an open source software and currently enjoys high popularity in the data processing community. Although Hadoop has been designed to run on a very large number of nodes (i.e., several thousand nodes), according to my observations, the software is also often used with significantly fewer machines in current IaaS clouds.

The challenge for both frameworks consists of two abstract tasks: Given a set of random integer numbers, the first task is to determine the k smallest of those numbers. The second task subsequently is to calculate the average of these k smallest numbers. The job is a classic representative for a variety of data analysis jobs whose particular tasks

vary in their complexity and hardware demands. While the first task has to sort the entire data set and therefore can take advantage of large amounts of main memory and parallel execution, the second aggregation task requires almost no main memory and, at least eventually, cannot be parallelized.

We implemented the described sort/aggregate task for three different experiments. For the first experiment, we implemented the task as a sequence of MapReduce programs and executed it using Hadoop on a fixed set of VMs. For the second experiment, we reused the same MapReduce programs as in the first experiment but devised a special MapReduce wrapper to make these programs run on top of Nephele. The goal of this experiment was to illustrate the benefits of dynamic resource allocation/deallocation while still maintaining the MapReduce processing pattern. Finally, as the third experiment, we discarded the MapReduce pattern and implemented the task based on a DAG to also highlight the advantages of using heterogeneous VMs.

For all three experiments, we chose the data set size to be 100 GB and generated the integer numbers according to the rules of the Jim Gray sort benchmark [101]. Consequently, each integer number had the size of 100 bytes and the data set contained about 10^9 distinct numbers. The cut-off variable k was set to 2×10^8, so the smallest 20% of all numbers had to be determined and aggregated. In order to make the data accessible to Hadoop, we started an HDFS [124] data node on each of the allocated VMs prior to the processing job and distributed the data evenly among the nodes. Since this initial setup procedure was necessary for all three experiments (Hadoop and Nephele), I will ignore it in the following performance discussion.

All three experiments were conducted on our local IaaS cloud. To manage the cloud and provision VMs on request of Nephele, we set up Eucalyptus [99]. Similar to Amazon EC2, Eucalyptus offers a predefined set of VM types a user can choose from. During our experiments, we used two different VM types: The first VM type was “m1.small” which corresponds to a machine with one CPU core, one GB of RAM, and a 128 GB disk. The second VM type, “c1.xlarge”, represents a machine with eight CPU cores, 18 GB RAM, and a 512 GB disk. Amazon EC2 has defined comparable VM types and offered them at a price of about 0.10 $, or 0.80 $ per hour (as of September 2009), respectively. Further details on the system configuration during the experiments can be found in the appendix.

3.4.1. Experiment 1: MapReduce and Hadoop

In order to execute the described sort/aggregate task with Hadoop, we created three different MapReduce programs which were executed consecutively.

The first MapReduce job reads the entire input data set, sorts the contained integer numbers ascendingly, and writes them back to Hadoop's HDFS file system. Since the MapReduce engine is internally designed to sort the incoming data between the map and the reduce phase, we did not have to provide custom map and reduce functions here. Instead, we simply used the TeraSort code, which has recently been recognized for being well-suited for these kinds of tasks [101]. The result of this first MapReduce job was a set of files containing sorted integer numbers. Concatenating these files yielded the fully sorted sequence of 10^9 numbers.

The second and third MapReduce jobs operated on the sorted data set and performed the data aggregation. Thereby, the second MapReduce job selected the first output files from the preceding sort job which, just by their file size, had to contain the smallest 2×10^8 numbers of the initial data set. The map function was fed with the selected files and emitted the first 2×10^8 numbers to the reducer. In order to enable parallelization in the reduce phase, we chose the intermediate keys for the reducer randomly from a predefined set of keys. These keys ensured that the emitted numbers were distributed evenly among the n reducers in the system. Each reducer then calculated the average of the received $\frac{2\times10^8}{n}$ integer numbers. The third MapReduce job finally read the n intermediate average values and aggregated them to a single overall average.

Since Hadoop is not designed to deal with heterogeneous compute nodes, we allocated six VMs of type "c1.xlarge" for the experiment. All of these VMs were assigned to Hadoop throughout the entire duration of the experiment.

We configured Hadoop to perform best for the first, computationally most expensive, MapReduce job: In accordance to [101] we set the number of map tasks per job to 48 (one map task per CPU core) and the number of reducers to 12. The memory heap of each map task as well as the in-memory file system were increased to one GB and 512 MB, respectively, in order to avoid unnecessarily spilling transient data to disk.

3.4.2. Experiment 2: MapReduce and Nephele

For the second experiment, we reused the three MapReduce programs we had written for the previously described Hadoop experiment and executed them on top of Nephele. In order to do so, we had to develop a set of wrapper classes providing limited interface compatibility with Hadoop and sort/merge functionality. These wrapper classes allowed us to run the unmodified Hadoop MapReduce programs with Nephele. As a result, the data flow was controlled by the executed MapReduce programs while Nephele was able to govern the VM allocation/deallocation and the assignment of tasks to VMs during the experiment. We devised this experiment in order to highlight the effects of the dynamic

resource allocation/deallocation while still maintaining comparability to Hadoop as well as possible.

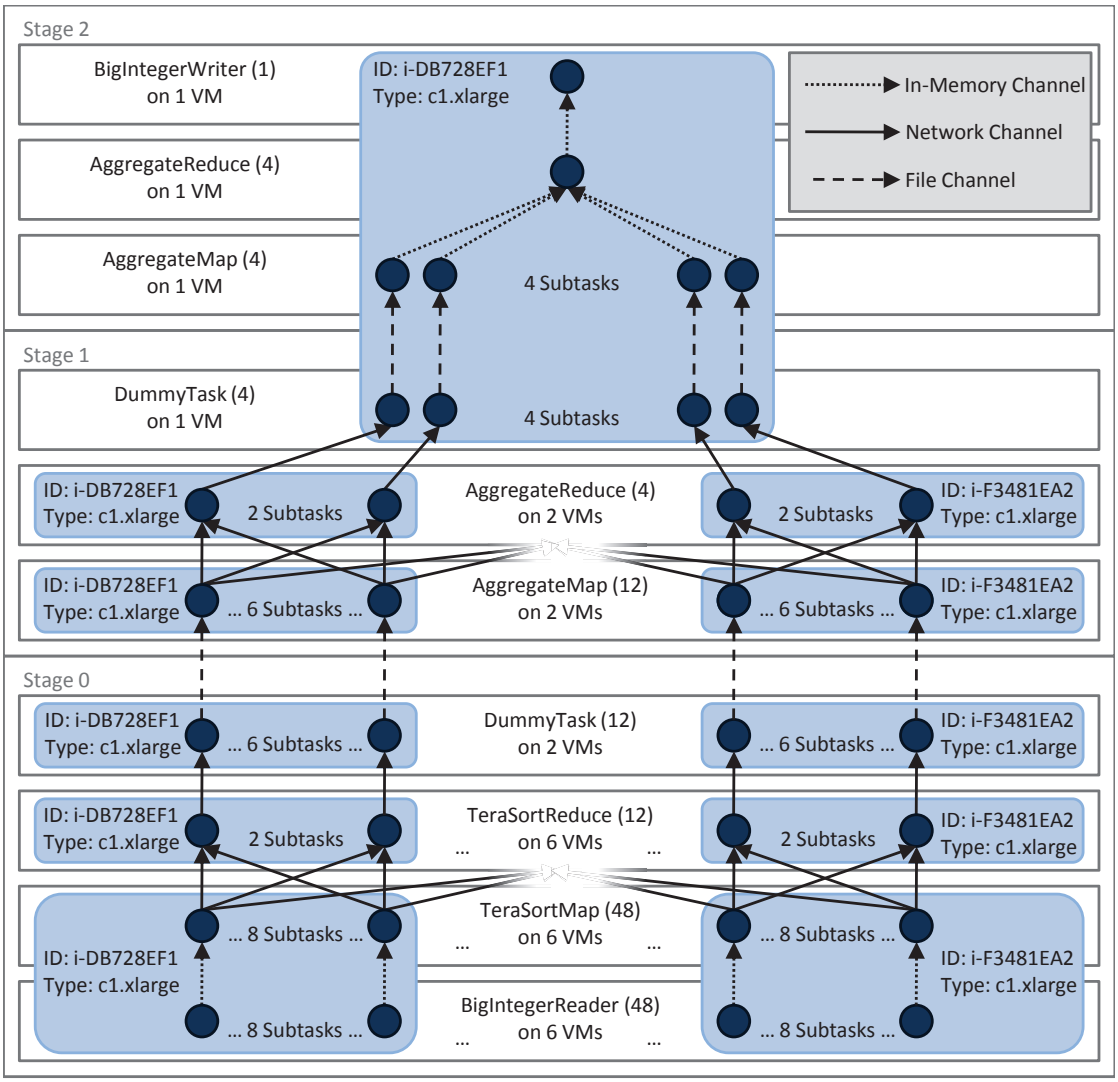


Figure 3.5.: The Execution Graph for Experiment 2 (MapReduce and Nephele).

Figure 3.5 illustrates the Execution Graph, we instructed Nephele to create so that the communication paths match the MapReduce processing pattern. For brevity, I omit a discussion on the original Job Graph. Following our experiences with the Hadoop experiment, we pursued the overall idea to also start with a homogeneous set of six "c1.xlarge" VMs, but to reduce the number of allocated VMs in the course of the experiment according to the previously observed workload. For this reason, the sort operation should

be carried out on all six VMs, while the first and second aggregation operation should only be assigned to two machines and to one machine, respectively.

The experiment's Execution Graph consisted of three Execution Stages. Each stage contained the tasks required by the corresponding MapReduce program. As stages can only be crossed via file channels, all intermediate data which occurred between two succeeding MapReduce jobs was completely written to disk, just like in the previous Hadoop experiment.

The first stage (Stage 0) included four different tasks, split into several different groups of subtasks and assigned to different VMs. The first task, BigIntegerReader, processed the assigned input files and emitted each integer number as a separate record. The tasks TeraSortMap and TeraSortReduce encapsulated the TeraSort MapReduce code which had been executed by Hadoop's mapper and reducer threads before. In order to meet the setup of the previous experiment, we split the TeraSortMap and TeraSortReduce tasks into 48 and 12 subtasks, respectively, and assigned them to six VMs of type "c1.xlarge". Furthermore, we instructed Nephele to construct network channels between each pair of TeraSortMap and TeraSortReduce subtasks. For the sake of legibility, only few of the resulting channels are depicted in Figure 3.5.

Although Nephele only maintains at most one physical TCP connection between two machines, we devised the following optimization: If two subtasks that are connected by a network channel are scheduled to run on the same VM, their network channel is dynamically converted into an in-memory channel. That way we were able to avoid unnecessary data serialization and the resulting processing overhead. For the given MapReduce communication pattern this optimization accounts for approximately 20% less network channels.

The records emitted by the BigIntegerReader subtasks were received by the TeraSortMap subtasks. The TeraSort partitioning function, located in the TeraSortMap task, then determined which TeraSortReduce subtask was responsible for the received record, depending on its value. Before being sent to the respective reducer, the records were collected in buffers of approximately 44 MB size and were presorted in memory. Considering that each TeraSortMap subtask was connected to 12 TeraSortReduce subtasks, this added up to a buffer size of 574 MB, similar to the size of the in-memory file system we had used for the Hadoop experiment previously.

Each TeraSortReducer subtask had an in-memory buffer of about 512 MB size, too. The buffer was used to mitigate hard drive access when storing the incoming sets of presorted records from the mappers. Like Hadoop, we started merging the first received presorted record sets using separate background threads while the data transfer from the mapper tasks was still in progress. In order to improve the overall performance, the final merge

step, resulting in one fully sorted set of records, was directly streamed to the next task in the processing chain.

The task DummyTask, the forth task in the first Execution Stage, simply emitted every record it received. It was used to direct the output of a preceding task to a particular subset of allocated VMs. Following the overall idea of this experiment, we used the DummyTask task in the first stage to transmit the sorted output of the 12 TeraSortReduce subtasks to the two instances Nephele would continue to work with in the second Execution Stage. For the DummyTask subtasks in the first stage (Stage 0), we shuffled the assignment of subtasks to VMs in a way that both remaining VMs received a fairly even fraction of the 2×10^8 smallest numbers. Without the shuffle, the 2×10^8 would all be stored on only one of the remaining VMs with high probability.

In the second and third stage (Stage 1 and 2 in Figure 3.5), we ran the two aggregation steps corresponding to the second and third MapReduce program in the previous Hadoop experiment. The two tasks AggregateMap and AggregateReduce thereby encapsulated the respective Hadoop code.

The first aggregation step was distributed across 12 AggregateMap and four AggregateReduce subtasks, which were assigned to the two remaining "c1.xlarge" VMs. In order to determine how many records each AggregateMap subtask had to process so that in total only the 2×10^8 numbers would be emitted to the reducers, we had to develop a small utility program. This utility program consisted of two components. The first component ran in the DummyTask subtasks of the preceding Stage 0. It wrote the number of records each DummyTask subtask had eventually emitted to a network file system share which was accessible to every VM. The second component, integrated in the AggregateMap subtasks, read those numbers and calculated what fraction of the sorted data was assigned to the respective mapper. In the previous Hadoop experiment this auxiliary program was unnecessary because Hadoop wrote the output of each MapReduce job back to HDFS anyway.

After the first aggregation step, we again used the DummyTask task to transmit the intermediate results to the last VM which executed the final aggregation in the third stage. The final aggregation was carried out by four AggregateMap subtasks and one AggregateReduce subtask. Eventually, we used one subtask of BigIntegerWriter to write the final result record back to HDFS.

3.4.3. Experiment 3: DAG and Nephele

In this third experiment, we were no longer bound to the MapReduce processing pattern. Instead, we implemented the sort/aggregation problem as a DAG and tried to exploit Nephele's ability to manage heterogeneous computing resources.

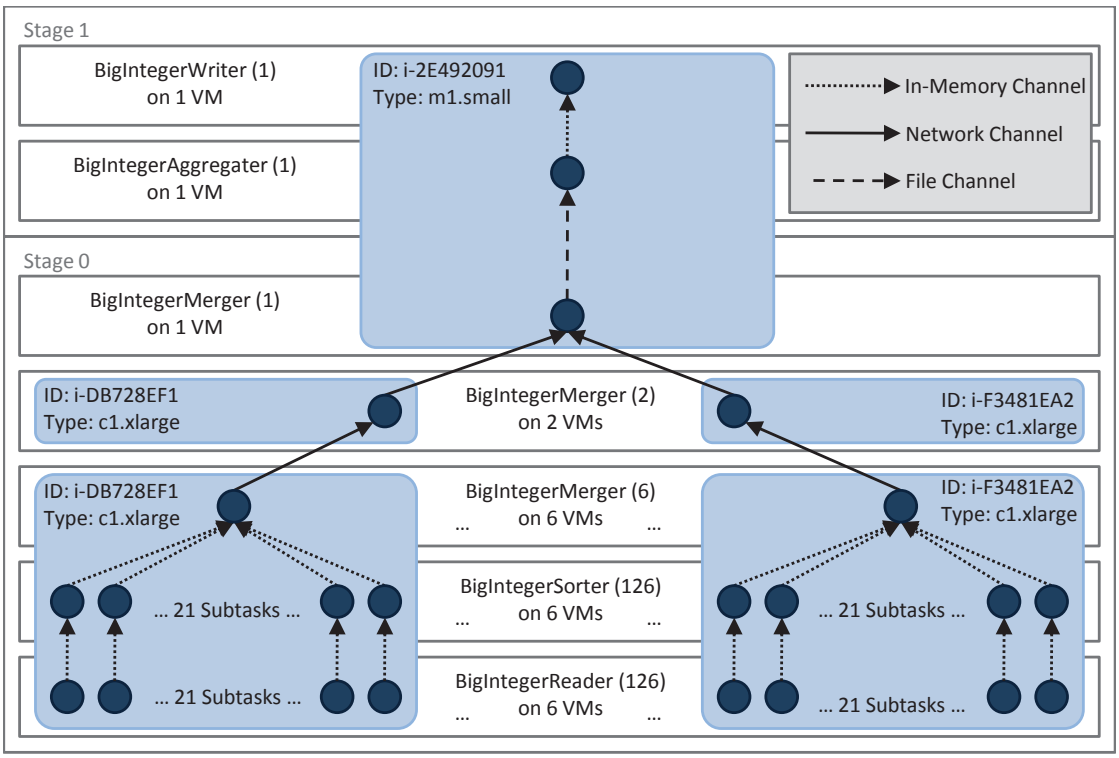


Figure 3.6.: The Execution Graph for Experiment 3 (DAG and Nephele).

Figure 3.6 illustrates the Execution Graph we instructed Nephele to create for this experiment. For brevity, I again leave out a discussion on the original Job Graph. Similar to the previous experiment, we pursued the idea that several powerful but expensive VMs are used to determine the 2×10^8 smallest integer numbers in parallel, while, after that, a single inexpensive VM is utilized for the final aggregation. The graph contained five distinct tasks, again split into different groups of subtasks. However, in contrast to the previous experiment, this one also involved VMs of different types.

In order to feed the initial data from HDFS into Nephele, we reused the BigIntegerReader task. The records emitted by the BigIntegerReader subtasks were received by the second task, BigIntegerSorter, which attempted to buffer all incoming records into main memory. Once it had received all designated records, it performed an in-memory quick sort and subsequently continued to emit the records in an order-preserving manner. Since the BigIntegerSorter task requires large amounts of main memory we split it into 126 subtasks and assigned these evenly to six instances of type "c1.xlarge". The preceding BigIntegerReader task was also split into 126 subtasks and set up to emit records via in-memory channels.

The third task, BigIntegerMerger, received records from multiple input channels. Once it has read a record from all available input channels, it sorts the records locally and always emits the smallest number. The BigIntegerMerger tasks occurred three times in a row in the Execution Graph. The first time it was split into six subtasks, one subtask assigned to each of the six "c1.xlarge" VMs. The second time the BigIntegerMerger task occurred in the Execution Graph, it was split into two subtasks. These two subtasks were assigned to two of the previously used "c1.xlarge" VMs. The third occurrence of the task was assigned to the new VMs of the type "m1.small".

Since we abandoned the MapReduce processing pattern, we were able to better exploit Nephele's streaming pipelining characteristics in this experiment. Consequently, each of the merge subtasks was configured to stop execution after having emitted 2×10^8 records. The stop command was propagated to all preceding subtasks of the processing chain, which allowed the Execution Stage to be interrupted as soon as the final merge subtask had emitted the 2×10^8 smallest records.

The fourth task, BigIntegerAggregater, read the incoming records from its input channels and summed them up. It was also assigned to the single "m1.small" VM. Since we no longer required the six "c1.xlarge" machines to run once the final merge subtask had determined the 2×10^8 smallest numbers, we changed the communication channel between the final BigIntegerMerger and BigIntegerAggregater subtask to a file channel. That way Nephele pushed the aggregation into the next Execution Stage and was able to deallocate the expensive VMs.

Finally, the fifth task, BigIntegerWriter, eventually received the calculated average of the 2×10^8 integer numbers and wrote the value back to HDFS.

3.4.4. Results

Figure 3.7, Figure 3.8, and Figure 3.9 show the performance results of our three experiment, respectively. All three plots illustrate the average VM utilization over time, i.e., the average utilization of all CPU cores in all VMs allocated for the job at the given point in time. The utilization of each machine has been monitored with the Unix command `top`. As in the previous chapter, it is broken down into the amount of time the CPU cores spent running the respective data processing framework (USR), the kernel and its processes (SYS), handling hardware (HIRQ) or software interrupts (SIRQ), and the time waiting for I/O to complete (WAIT). To illustrate the impact of network communication, the plots additionally show the average amount of IP traffic flowing between the VMs over time.

I begin with discussing Experiment 1 (MapReduce and Hadoop): For the first MapReduce job, TeraSort, Figure 3.7 shows a fair resource utilization. During the map (point

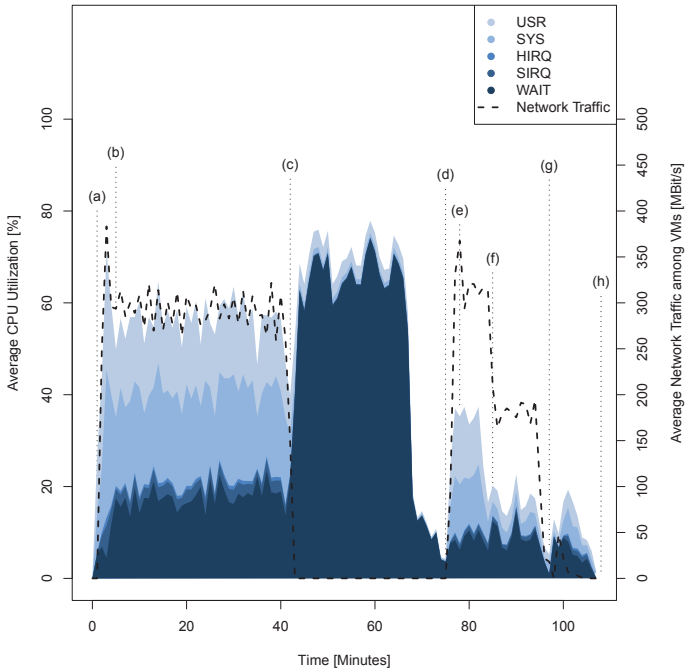


Figure 3.7.: Results of Experiment 1 (MapReduce and Hadoop).

(a) to (c)) and reduce phase (point (b) to (d)) the overall system utilization ranges from 60 to 80%. This is reasonable since we configured Hadoop's MapReduce engine to perform best for this kind of task. For the following two MapReduce jobs, however, the allocated VMs are oversized: The second job, whose map and reduce phases range from point (d) to (f) and point (e) to (g), respectively, can only utilize about one third of the available CPU capacity. The third job (running between point (g) and (h)) can only consume about 10% of the overall resources.

The reason for Hadoop's eventual poor VM utilization is its assumption to run on a static compute cluster. Once the MapReduce engine has been started on a set of VMs, no machine can be removed from that set without the risk of losing important intermediate results. As in this case, all six expensive VMs must be allocated throughout the entire experiment and unnecessarily contribute to the processing cost.

Figure 3.8 shows the system utilization for executing the same MapReduce programs on top of Nephele. For the first Execution Stage, corresponding to the TeraSort map and reduce tasks, the overall resource utilization is comparable to the one of the Hadoop experiment. During the map phase (point (a) to (c)) and the reduce phase (point (b) to (d)) all six "c1.xlarge" machines show an average utilization of about 80%. However,

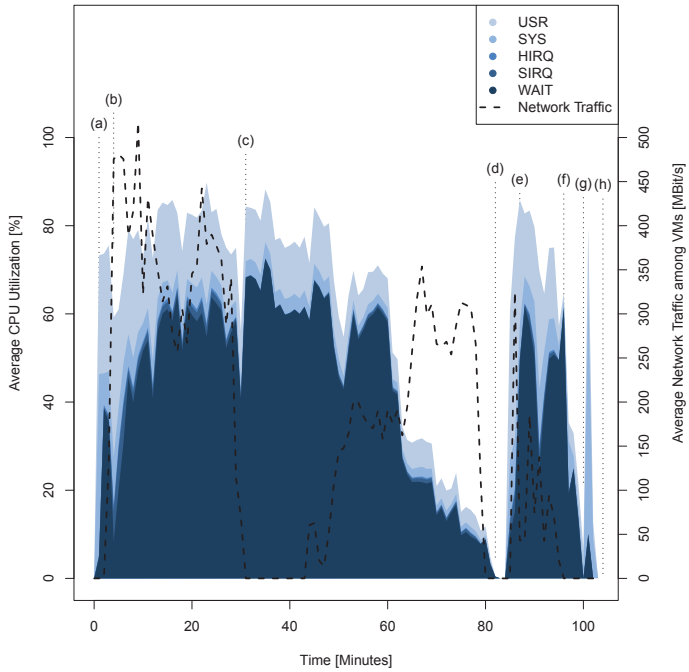


Figure 3.8.: Results of Experiment 2 (MapReduce and Nephele).

after approximately 42 minutes, Nephele starts transmitting the sorted output stream of each of the 12 TeraSortReduce subtasks to the two VMs which are scheduled to remain allocated for the upcoming Execution Stages. At the end of Stage 0 (point (d)), Nephele is aware that four of the six "c1.xlarge" machines are no longer required for the upcoming computations and deallocates them.

Since the four deallocated machines do no longer contribute to the number of available CPU cores in the second stage, the remaining VMs again match the computational demands of the first aggregation step. During the execution of the 12 AggregateMap subtasks (point (d) to (f)) and the four AggregateReduce subtasks (point (e) to (g)), the utilization of the allocated VMs is about 80%. The same applies to the final aggregation in the third Execution Stage (point (g) to (h)) which is only executed on one allocated "c1.xlarge" machine.

Finally, I want to discuss the results of the third experiment (DAG and Nephele) as depicted in Figure 3.9: At point (a) Nephele has successfully allocated all VMs required to start the first Execution Stage. Initially, the BigIntegerReader subtasks begin to read their splits of the input data set and emit the created records to the BigIntegerSorter subtasks. At point (b) the first BigIntegerSorter subtasks switch from buffering the

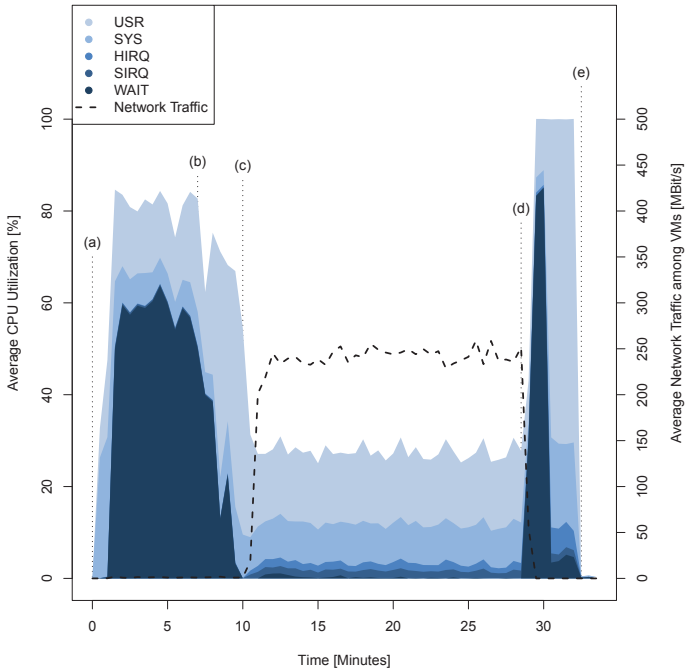


Figure 3.9.: Results of Experiment 3 (MapReduce and DAG).

incoming records to sorting them. Here, the advantage of Nephele's ability to assign specific VM types to specific kinds of tasks becomes apparent: Since the entire sorting can be done in main memory, it only takes several seconds. Three minutes later (c), the first BigIntegerMerger subtasks start to receive the presorted records and transmit them along the processing chain.

Until the end of the sort phase, Nephele can fully exploit the power of the six allocated "c1.xlarge" machines. After that period the computational power is no longer needed for the merge phase. From a cost perspective it is now desirable to deallocate the expensive machines as soon as possible. However, since they hold the presorted data sets, at least 20 GB of records must be transferred to the inexpensive "m1.small" VM first. Here, we identified the network to be the bottleneck, so much computational power remains unused during that transfer phase (from (c) to (d)). In general, this transfer penalty must be carefully considered when switching between different VM types during job execution. The adaptive compression algorithm, which is presented in Chapter 5 of this thesis, might have been beneficial in this case to trade CPU against I/O load [1]. However, in order to achieve an unbiased comparison to the other two experiments, the feature is

[1] Although the numbers generated according to the Jim Gray sort benchmark are considered to be random, they actually contain recurring patterns which make them suitable for compression.

not activated during these experiments.

At point (d), the final BigIntegerMerger subtask has emitted the 2×10^8 smallest integer records to the file channel and advises all preceding subtasks in the processing chain to stop execution. All subtasks of the first stage have now been successfully completed. As a result, Nephele automatically deallocates the six VMs of type "c1.xlarge" and continues the next Execution Stage with only one machine of type "m1.small" left. In that stage, the BigIntegerAggregater subtask reads the 2×10^8 smallest integer records from the file channel and calculates their average. Again, since the six expensive "c1.xlarge" nodes no longer contribute to the number of available CPU cores in that period, the processing power allocated from the cloud again fits the task to be completed. At point (e), after 33 minutes, Nephele has finished the entire processing job.

Considering the short processing times of the presented tasks and the fact that most cloud providers offer to lease a VM for at least one hour, I am aware that Nephele's savings in time and cost might appear marginal at first glance. However, I want to point out that these savings grow by the size of the input data set. Due to the size of our test cloud we were forced to restrict data set size to 100 GB. For larger data sets, more complex processing jobs become feasible, which also promises more significant savings.

3.5. Related Work

In recent years a variety of MTC or DISC frameworks have been developed. Although these systems typically share similar goals (e.g. simplifying the deployment of jobs across a large number of compute nodes, hiding issues of parallelism, or fault tolerance), they aim at different fields of application.

MapReduce [43] (or the open source version Hadoop [124]) is designed to run data analysis jobs on a large amount of data, which is expected to be stored across a large set of share-nothing commodity servers. MapReduce is highlighted by its simplicity: Once a user has fit his program into the required map and reduce pattern, the execution framework takes care of splitting the job into subtasks, distributing and executing them. A single MapReduce job always consists of a distinct map and possibly a reduce program. However, a variety of projects exists to coordinate the execution of a sequence of MapReduce jobs [100, 126, 125].

Since the publication of the original MapReduce research paper by Dean and Ghemawat, several alternative implementations of the programming pattern have surfaced. While some aim at improving the performance for particular types of MapReduce jobs [45], others extend the original framework by additional features, such as pipelining [42]. However, MapReduce and those derivatives have been clearly designed for large static

clusters. Although they can deal with sporadic node failures, the available computing resources are essentially considered to be a fixed set of homogeneous machines.

The Pegasus framework by Deelman et al. [44] has been designed for mapping complex scientific workflows onto grid systems. Similar to Nepehle, Pegasus lets its users describe their jobs as a DAG with vertices representing the tasks to be processed and edges representing the dependencies between them. The created workflows remain abstract until Pegasus creates the mapping between the given tasks and the concrete computing resources available at runtime. The authors incorporate interesting aspects like the scheduling horizon which determines at what point in time a task of the overall processing job should apply for a computing resource. This is related to the stage concept in Nephele. However, Nephele's stage concept is designed to minimize the number of allocated instances in the cloud and clearly focuses on reducing cost. In contrast, Pegasus' scheduling horizon is used to deal with unexpected changes in the execution environment. Pegasus uses DAGMan and Condor-G [57] as its execution engine. As a result, different tasks can only exchange data via files.

Thao et al. introduced Swift [151] to reduce the management issues which occur when a job involving numerous tasks has to be executed on a large, possibly unstructured, set of data. Building upon components like CoG Karajan [133], Falkon [107], and Globus [54], the authors present a scripting language which allows to create mappings between logical and physical data structures and to conveniently assign tasks to these.

The system our approach probably shares most similarities with is Dryad [74]. Dryad also runs DAG-based jobs and offers to connect the involved tasks through either file, network, or in-memory channels. However, it assumes an execution environment which consists of a fixed set of homogeneous worker nodes. The Dryad scheduler is designed to distribute tasks across the available compute nodes in a way that optimizes the throughput of the overall cluster. It does not include the notion of processing cost for particular jobs.

Borkar et al. demonstrated a framework for data-intensive distributed computing called Hyracks [33]. Similar to Nephele and Dryad, Hyracks also describes processing jobs as DAGs. In Hyracks, the vertices of the DAG implement typical data processing operators like, e.g., readers/writers, sorters, or joiners. The DAG's edges represent so-called connectors between these operators. Similar to Dryad, Hyracks also has a strong focus on clusters of commodity computers and does not address dynamic resource allocation.

In terms of on-demand resource provisioning several projects have arisen recently: Dornemann et al. [46] presented an approach to handle peak-load situations in BPEL workflows using Amazon EC2. Ramakrishnan et al. [108] discussed how to provide a uniform resource abstraction over grid and cloud resources for scientific workflows. Both

projects rather aim at batch-driven workflows than the data-intensive, pipelined workflows Nephele focuses on. The FOS project [140] has recently presented an operating system for multicore and clouds which is also capable of on-demand VM allocation.

3.6. Summary

In this chapter the thesis discussed design fundamentals data processing frameworks must meet in order to be able to exploit the dynamic resource allocation offered by today's IaaS platforms. Based on these design fundamentals I presented Nephele as the first parallel data processing framework to explicitly incorporate the cloud's ability for rapid resource provisioning and heterogeneous computing resources.

Unlike existing frameworks for massively parallel data processing, Nephele no longer assumes that it owns the available compute nodes and that it can freely dispose of them. Instead, the system follows the idea that the available computing resources are potentially only leased for a particular time span and may incur monetary cost. With this idea in mind, the chapter outlined Nephele's basic architecture and programming abstraction. In particular, it described how users can compose complex processing jobs from individual tasks and influence the job's scheduling and execution through a set of possible annotations.

With respect to dynamic resource allocation, this chapter discussed several strategies for VM allocation/deallocation in the course of a processing job and pointed out existing scheduling tradeoffs between preserving intermediate results in case of temporary resource unavailabilities and potentially shorter lease durations. Finally, Nephele's performance was evaluated by means of a MapReduce-inspired data analysis job. In the scope of the experiments, I highlighted the benefits of both dynamic resource allocation as well as resource heterogeneity through a comparison with the popular data processing framework Hadoop based on resource utilization, processing time, and cost.

The chapter also identified directions for further research. In particular, generating efficient execution plans in the absence of the user's job annotations is currently a challenging subject. The thesis will further investigate the problem of finding reasonable degrees of parallelization for a processing job in the next chapter. Moreover, Chapter 7 will address optimization opportunities that arise from combining Nephele with semantically richer, higher level programming abstractions.

Since Nephele's programming abstraction is very general and allows its users to express a broad range of large-scale data analysis problems, I will use Nephele as a representative of a parallel data processing framework for the remaining chapters of this thesis.

4. Detecting Bottlenecks in Parallel Data Flow Programs

Contents

4.1. Processing Model and Problem Definition 54

4.1.1. Processing Model 55

4.1.2. Problem Definition 56

4.2. Bottleneck Detection Algorithms 57

4.3. Implementation in Nephele 60

4.4. Evaluation 62

4.4.1. Use Case 63

4.4.2. Results 64

4.5. Related Work 67

4.6. Summary 69

Since modern IaaS clouds suggest access to a virtually unlimited pool of computing resources, this also stresses the problem of finding a reasonable degree of parallelism for data processing jobs executed on top of these platforms. While in a classic cluster environment the size of the cluster (i.e., the number of CPU cores or the amount of main memory) has traditionally provided an upper bound for possible scale-outs, the dimensions of commercial cloud data centers together with the cloud's cost model has rendered this limit obsolete. Instead, researchers might be tempted to strive for shorter completion times by increasing their jobs' level of parallelization.

However, for the vast majority of MTC-like applications, the range of sensible scale-outs is limited. Too large degrees of parallelism will eventually turn the cloud's underlying I/O infrastructure, which delivers the input data to the individual compute nodes, into a bottleneck. As a result, large parts of the rented VMs will remain idle and unnecessarily contribute to the processing cost. Contrary to that, too low degrees of parallelization might cause CPU bottlenecks in the processing chain so that successive tasks might suffer from a lack of input data. In this case, the VMs of these successive tasks will also be underutilized and diminish the job's cost efficiency.

The challenge of finding sensible scale-outs carries special weight for processing jobs which contain arbitrary user code. These kinds of processing jobs typically only allow very few assumptions about the job's concrete computational characteristics in the forefront of its execution. Moreover, many of those jobs are also executed in a producer-consumer fashion [74, 125, 24] so that each of the job's tasks depends on sufficient amounts of input data from its predecessors in order to reach its optimal throughput and a high system utilization. As a result, in these cases, the problem is not only to detect the sweet spot in the range of possible scale-outs for the job itself but for each of its tasks individually with respect to its predecessors and successors.

Being designed for classic cluster setups, existing data processing frameworks only offer very little support for this problem. Current tutorials on this topic [123] mainly propose back-of-the-envelope calculations and ignore the characteristics of the job. In particular, these tutorials do not address the problem of adjusting the scale-out of interdependent tasks with potentially very different computational complexities.

To mitigate this current lack of assistance, this chapter presents a scheme to detect CPU and I/O bottlenecks in DAG-based data flow programs at runtime [27]. As explained in the previous chapter, the detection of these bottlenecks represents an important prerequisite for manually or automatically scaling out these kinds of programs in order to increase their processing performance and cost efficiency.

Although I will demonstrate the scheme based on the Nephele framework, it is in principle applicable to any parallel data processing framework which describes jobs as DAGs and follows a classic producer-consumer pattern. Therefore, this chapter will start with an abstract introduction of the assumed processing model and a problem definition. Based on this foundation, it will introduce the algorithms for bottleneck detection and the concrete implementation in Nephele. Finally, I will illustrate the applicability of our approach through an experimental evaluation.

4.1. Processing Model and Problem Definition

This section will explain the general processing model the bottleneck detection algorithm later bases on. Moreover, it will explain CPU and I/O bottlenecks with respect to parallel DAG-based data flow programs and define the concrete problem which is addressed by the presented algorithm.

Note that with regard to the potentially false display of system characteristics within a VM (cf. Section 2.3), we deliberately decided not to include any physical hardware figures (such as maximum/available network throughput) into the processing model.

Instead, the processing model will only rely on metrics that can be directly derived from the producer-consumer pattern.

4.1.1. Processing Model

The bottleneck detection approach presented in this chapter can be applied to arbitrary MTC-like processing jobs which fulfill the assumptions described in the following:

- **Assumption 1:** The processing job can be modeled as a DAG $G = (V_G, E_G)$. Therein, each vertex $v \in V_G$ of the DAG represents a separate *task* of the overall processing job. The directed edges $e \in E_G$ between the vertices model the *communication channels* through which data is passed on from one task to the next. In the context of the Nephele framework, which has been introduced in the previous chapter, Nephele's Job Graph would be an example of such a DAG.
- **Assumption 2:** The interaction between the individual tasks of the processing job follows a producer-consumer pattern. Tasks exchange data through communication channels in distinct units. I will comply with the Nephele terminology and also refer to these units as *records* in this chapter. All communication between tasks takes place through communication channels modeled in the DAG.
- **Assumption 3:** A communication channel is unidirectional and follows the abstraction of a FIFO queue. Moreover, it is backed by a buffer which can temporarily store a particular number of records. The concrete size of the buffer is allowed to change over time. This characteristic is motivated by the varying window size of a TCP connection which can act as the physical foundation of the communication channel in practice. However, the buffer size is expected to be eventually constrained by an arbitrary but fixed value. I will refer to the current buffer size as the channel's capacity. Any attempts to write records to a channel beyond its capacity limit will cause the producing task to be blocked until the consuming task has removed at least one record from the channel. Analogously, any attempt to read from a channel which currently does not hold any records will cause the consuming task to be blocked until at least one record is written to the channel.
- **Assumption 4:** Each task of the DAG consists of sequential and deterministic, yet potentially unknown user code. It can be parallelized so that multiple instances of the same task operate on different fractions of the task's overall input data. I will also stick to the terminology introduced in the previous chapter and refer to such a parallel instance of a task as a *subtask*.
- **Assumption 5:** At runtime, each subtask is in one of the following states: PROCESSING or WAITING. A state change is always triggered by one of its connected

communication channels. A subtask is in state WAITING when it is either waiting to write records to an outgoing channel or waiting for records to arrive from an incoming channel; otherwise it is in state PROCESSING. Hence, if sufficient input records are available and capacity is left to write out the result records, a subtask will not enter the WAITING state. The current state of a subtask can be accessed at anytime during its execution. Note that even waiting for I/O other than communication channel I/O (such as reading or writing from/to hard disk) is therefore considered as processing time.

- **Assumption 6:** At runtime, each communication channel is either in the state SATURATED when its buffer's capacity limit has been reached; otherwise it is in state READY. Similar to the tasks, the current state of a channel is assumed to be accessible at anytime throughout its lifetime.
- **Assumption 7:** If a specific record is processed by the same task in different job executions, the performance characteristics (processing complexity, value and size of produced output) remain the same. This assumption allows profiling a job and using the gained knowledge to improve a second execution of the job.

4.1.2. Problem Definition

After having explained the general processing model, the section will continue explaining our understanding of CPU and I/O bottlenecks. Intuitively, the different types of bottlenecks can be described as follows:

- **CPU bottlenecks** are tasks whose throughput is limited by the CPU resources they can utilize. CPU bottlenecks are distinguished by the fact that they have sufficient amounts of input data to process, however, subsequent tasks in the processing chain suffer from a lack thereof.
- **I/O bottlenecks** are those communication channels which are requested to transport more records per time unit on an average than the underlying transport infrastructure (for example network interconnects) can handle.

The problem this chapter addresses is the detection of such bottlenecks only based on the previously described task and channel states (PROCESSING/WAITING and READY/SATURATED). Through this abstraction the approach becomes independent of the concrete physical compute and communication resource, which may be hard to observe in (shared) virtualized environments like IaaS clouds.

It is important to point out that our bottleneck detection approach considers a job DAG on the task level. This means, we do not aim at detecting CPU or I/O bottlenecks

at individual subtasks. Bottlenecks on a subtask level typically indicate load balancing problems. They do not provide any clues about bottlenecks which stem from inappropriate levels of parallelization of distinct tasks, which is the primary intention of this approach. So although a task may be executed as hundreds or thousands of parallel subtasks, algorithmically it is treated as a single task. Section 4.3 will provide details on how the aggregation of the necessary data actually takes place in a distributed setup.

4.2. Bottleneck Detection Algorithms

According to our definition of CPU and I/O bottlenecks, the following algorithm is capable of detecting those bottlenecks in DAG-based parallel data flow programs. The algorithm is applicable to any parallel data processing framework whose jobs fit the model presented in Section 4.1. The algorithm is expected to be triggered periodically during the job execution in order to account for the fact that the behavior of the processing job's individual tasks may change over time and, hence, disclose different bottlenecks in the course of a job's execution.

Algorithm 1 illustrates the overall approach of our bottleneck detection algorithm. The algorithm is passed the DAG G which represents the currently executed job. Initially, the function $\mathit{ReverseTopologicalSort}(G)$ (line 1) creates and returns a list L_{RTS} with all vertices of G. The order of the vertices within the list corresponds to a reverse topological ordering, i.e., vertices with no outgoing edges appear first in the list.

Algorithm 1 DetectBottlenecks($G := (V_G, E_G)$)

1: $L_{RTS} \leftarrow \mathit{ReverseTopologicalSort}(G)$
2: **for all** v **in** L_{RTS} **do**
3: $\quad v.\mathit{isCpuBottleneck} \leftarrow \mathit{IsCpuBottleneck}(v, G)$
4: **end for**
5: **if** $\nexists v \in L_{RTS} : v.\mathit{isCpuBottleneck}$ **then**
6: $\quad$ **for all** v **in** L_{RTS} **do**
7: $\quad\quad E_v = \{(v, w) | w \in V_G \wedge (v, w) \in E_G\}$
8: $\quad\quad$ **for all** $e \in E_v$ **do**
9: $\quad\quad\quad e.\mathit{isIoBottleneck} \leftarrow \mathit{IsIoBottleneck}(e, G)$
10: $\quad\quad$ **end for**
11: $\quad$ **end for**
12: **end if**

The list L_{RTS} is then traversed from the beginning to the end. For each vertex v we check whether v is considered a CPU bottleneck. The particular properties for a vertex to meet the CPU bottleneck condition are checked within the function $\mathit{IsCpuBottleneck}(v, G)$

(line 3), which is explained later in this section. The result of the check is returned and stored in the Boolean variable $v.isCpuBottleneck$.

In order to detect I/O bottlenecks, we take a similar approach. Again, we traverse each vertex v of the job DAG G according to their reverse topological order. For each outgoing edge $e = (v, w)$ of v we check whether e meets the conditions of an I/O bottleneck. However, we only perform the check if no CPU bottleneck has been discovered before. The discussion of Algorithm 3 later in this section will clarify the necessity for this constraint.

Algorithm 2 describes how we check whether a particular vertex $v \in V_G$ is a CPU bottleneck. The algorithm checks for two conditions which must be fulfilled in order to classify v as a CPU bottleneck.

A crucial prerequisite for a CPU bottleneck is that the task represented by vertex v spends almost the entire CPU time given to it in the state PROCESSING. We introduce the function $pt(v)$ which is defined as the arithmetic mean of the fractions of time the subtasks (i.e., the different parallel instances of the task) spent in the state PROCESSING during the last time unit of their execution. Synchronization issues may cause individual subtasks to spend short periods of time in the state WAITING. For this reason we introduce a threshold α for $pt(v)$ which must be exceeded so that v is considered a bottleneck. In our practical experiments we found 90% to be a reasonable value for α.

Algorithm 2 IsCpuBottleneck(v, G)

1: **if** $pt(v) \leq \alpha$ **then**
2: **return** *FALSE*
3: **end if**
4: **if** $\exists s \in vsucc^*(v, G) : s.isCpuBottleneck$ **then**
5: **return** *FALSE*
6: **end if**
7: **return** *TRUE*

The second condition for a CPU bottleneck considers the set of v's successors, $vsucc^*(v, G)$, i.e., the vertices which can be reached from v. Formally, a vertex s is in $vsucc^*(v, G)$ if there exists a path $p = (v_1, ..., v_n)$ such that $v_1 = v$, $v_n = s$ and $(v_i, v_{i+1}) \in E_G$, $1 \leq i < n$. For each such successor s we check if s has been classified as a CPU bottleneck. The order in which we traverse the vertices in the job DAG G guarantees that the CPU bottleneck flag $s.isCpuBottleneck$ of all of vertex v's successors has been updated before the function *IsCpuBottleneck* is called with v itself.

The necessity for this second condition becomes apparent when recalling the definition of a CPU bottleneck from Section 4.1. According to that definition, a CPU bottleneck

is characterized by high CPU load and the fact that it provides successor vertices with insufficient amounts of input data. However, if any successor s of vertex v has already been identified as a CPU bottleneck, this would mean s does not suffer from insufficient amounts of input data because the amount of input data s receives is sufficient to max out its CPU time. As a result, classifying vertex v as a CPU bottleneck requires all of its successors not to be classified as CPU bottlenecks.

Algorithm 3 IsIoBottleneck($e := (v, w)$, G)

1: **if** $st(e) \leq \beta$ **then**
2: **return** *FALSE*
3: **end if**
4: **if** $\exists s \in esucc^*(v, G) : s.isIoBottleneck$ **then**
5: **return** *FALSE*
6: **end if**
7: **return** *TRUE*

After all CPU bottleneck flags have been updated, Algorithm 3 checks whether an edge should be considered an I/O bottleneck. For an edge $e = (v, w) \in E_G$, $st(e)$ denotes the arithmetic mean of the fractions of time the communication channels represented by the edge e spent in the state SATURATED during the last time unit of v's execution.

Similar to CPU bottlenecks, we consider two conditions for I/O bottlenecks. First, $st(e)$ must be above a threshold β which indicates that the communication edges represented by the edge e spent the majority of the considered time interval in the state SATURATED. In practice we found 90% to be a reasonable threshold for β, so temporary fluctuations in the channel utilization do not lead to wrong bottleneck decisions.

The second condition again considers the successors of an edge e. By $esucc^*(e, G)$ we denote the set of successor edges of e. Formally, an edge $s = (t, u)$ is in $esucc^*(e, G)$ if there exists a path $p = ((v_0, v_1), ..., (v_{n-1}, v_n))$ such that $(v_i, v_{i+1}) \in E_G$, $0 \leq i < n$ and $v_0 = v$, $v_1 = w$, $v_{n-1} = t$, and $v_n = u$. An edge e is only classified as an I/O bottleneck if no successor edge has been classified as an I/O bottleneck before. Again, the order in which we traverse the edges ensures the appropriate update of the bottleneck flags.

The I/O bottleneck approach bears some discussion. Generally, there exist two possible reasons for high values of $st(e)$. The first reason is that the maximum throughput rate of the underlying transport infrastructure which backs the communication channel has been reached. This corresponds to our definition of an I/O bottleneck in Section 4.1. The second possible, however spurious, reason is an insufficient consumption rate of the task which consumes data from e. This, in turn, can be caused by two circumstances: First, a CPU bottleneck in the DAG could affect the consumption rate of the respective task. However, since we only check for I/O bottlenecks if no CPU bottleneck has been

detected before, this cause can be eliminated. Second, another I/O bottleneck could exist in the remainder of the processing chain. Yet, this is impossible because of the second condition (line 4) of Algorithm 3.

4.3. Implementation in Nephele

A key requirement to apply the presented bottleneck detection algorithm to the Nephele data processing framework is the ability to derive the state information from the tasks and their communication channels at runtime. Therefore, we have devised a special profiling subsystem which continuously monitors each Nephele subtask during its execution and periodically calculates aggregates of the obtained data. These aggregates then represent the foundation for implementing the utilization functions $pt(v)$ and $st(e)$.

The major objective of the profiling subsystem is to constantly collect data on how much time a Nephele subtask spent in a particular processing state during the last time unit of its execution. On an operating system level, collecting such data for individual processes is easy. Linux, for example, offers the `/proc/` interfaces to obtain detailed statistics on the individual operating system processes and their corresponding states.

However, in order to facilitate fast memory-based communication between two tasks, Nephele cannot always map different tasks to different operating system processes. Instead, the usage of in-memory channels forces Nephele to instantiate different tasks as different threads within the same process. With respect to collecting the profiling information this means that we also have to be able to monitor the processing states of individual threads. Since most parts of Nephele are written in Java, there are several different options to achieve this goal.

Our first profiling approach was based on the Java Virtual Machine Tool Interface (JVMTI). JVMTI provides access to the internal state of the Java Virtual Machine (JVM). It allows writing so-called agents in a native language like C or C++, so unlike Java itself the profiling extension was platform dependent. The agent is then executed in the same process as the JVM and is notified about occurring events with the help of callback functions.

Our second profiling approach relied on the Java Management Extension (JMX). JMX comprises the MXBeans platform package which provides access to, among other things, the JVM's runtime, thread and memory subsystem. In particular, we used the class `ThreadMXBean` to determine the CPU time of individual threads.

In order to evaluate the impact of both profiling approaches on the actual subtask execution, we implemented both approaches and devised a CPU-intensive sample job. We executed the sample job several times without the profiling component as well as with

the JVMTI or JMX-based profiling component enabled. Each version of the profiling component queried the information on the monitored thread's CPU time every second. The results of the comparison are depicted in Figure 4.1.

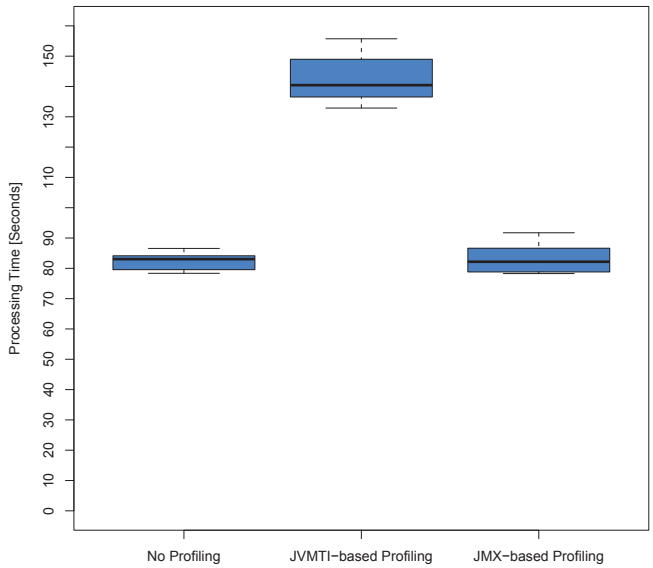


Figure 4.1.: Profiling overhead using the JVMTI- and JMX-based approach.

Without profiling the mean execution time of the sample job was around 82 seconds. The JMX-based profiling component proved to be very lightweight. It only increased the mean execution time by less than 1%. In contrast to that, the JVMTI-based component led to a significant drop in execution speed. On average the subtasks' completion time was increased by almost 74%. A closer examination revealed that the frequent calls of the native agent code were the main reason for the performance penalty.

As a result of the first performance evaluation, we implemented the functions $pt(v)$ and $st(e)$ based on the lightweight JMX approach. In order to generate the values of $pt(v)$ we query the JMX interface every five seconds for statistics on every subtask thread. The statistics let us derive detailed information on the distribution of the thread's CPU time among the different internal states of the JVM. These internal states are:

- **USR:** The amount of time the monitored task thread was executed in user mode.
- **SYS:** The amount of time the monitored task thread was executed in system mode.

- **BLK:** The amount of time the monitored task thread was blocked because of mutual exclusion.
- **WAIT:** The amount of time the monitored task thread was intentionally instructed to wait.

Since the threads of Nephele subtasks only enter the WAIT state as a result of congested or starved communication channels, we can map this state directly to the WAITING state of our bottleneck algorithm. The other three internal JVM states (USR, SYS, BLK) are mapped to the PROCESSING state. This also complies with Assumption 5 of Section 4.1.

In order to determine the utilization of communication channels, we simply store a timestamp for the respective channel whenever a subtask thread either attempts to read from or write to the channel and the attempt leads to the subtask thread switching its state from PROCESSING to WAITING. Moreover, we store a timestamp whenever new incoming data from the considered channel or the completed transmission of outgoing data allow the subtask to switch from state WAITING back to the state PROCESSING (cf. Assumption 5). Based on these timestamps we can then calculate how much time the channel spent in the states SATURATED and READY.

After having calculated the utilization statistics for each Nephele subtask locally at the respective Task Managers, the profiling data is forwarded to Nephele's central management component, the Job Manager. The Job Manager then calculates an arithmetic mean of the individual subtask statistics. During the experiments on our mid-size cloud testbed, the amount of data that was generated by our profiling subsystem was negligibly small and did not account for any observable load on the Job Manager. In larger setups, when scalability might become a bigger concern, the profiling overhead can be reduced by increasing the reporting interval.

4.4. Evaluation

This section will evaluate the effectiveness of the bottleneck detection approach with the help of a concrete use case. The use case picks up the optimization idea presented in Section 3.3: A developer intends to use an IaaS cloud to repeatedly run an MTC-like processing job. The job consists of several individual tasks that interact with each other. The developer strives to finish the processing job as fast as possible without paying for unutilized computing resources. Therefore, he uses the feedback provided by the bottleneck detection algorithm to gradually adjust each task's degree of parallelism before the begin of the job's next run.

4.4.1. Use Case

The job we devised for our use case is inspired by the famous Hadoop job of the New York Times, which was used to convert their four TB large article archive from TIFF images to PDF using 100 VMs on Amazon EC2 [61].

In our case the conversion job consists of six distinct tasks. Figure 4.2 depicts the corresponding Nephele Job Graph. The first task, *File Reader*, initially reads the individual image files from disk and sends each image as a separate record to the second task, *Optical Character Recognition (OCR) Task*. The OCR Task then applies a text recognition algorithm to the received image. The result of the text recognition, a regular string, is then processed in a twofold manner. First, the recognized text pattern of the image is passed to the task *PDF Creator*. Second, each word within the text pattern is forwarded to a task *Inverted Index Task* together with the name of the original image file.

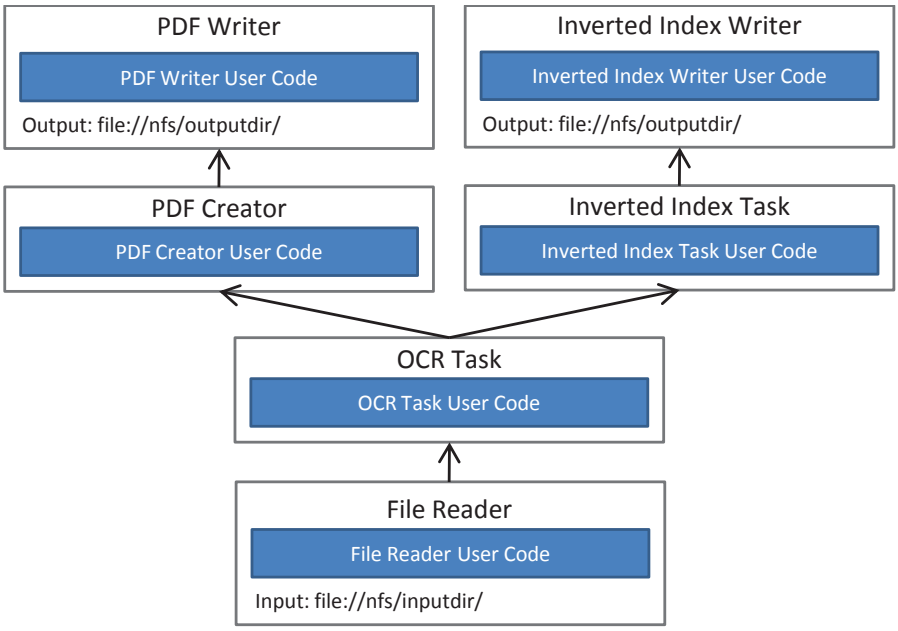


Figure 4.2.: The Nephele Job Graph used for the evaluation.

The task PDF Creator receives the recognized text pattern of each original image as a separate record. The text pattern is then converted to a PDF document and emitted to the task *PDF Writer*. The PDF Writer task eventually writes the received PDF document back to disk.

The Inverted Index Task receives tuples of words from the recognized text patterns and the names of the originating image files. As its name implies, the task uses the received

tuples to construct an inverted index. This inverted index can later be used to facilitate a keyword search on the created PDF file set. In our evaluation the created index has been small enough to fit into the node's main memory. After having received all input tuples, the task sends out tuples of each indexed word together with a list of filenames in which the word occurs to the task *Inverted Index Writer*. Inverted Index Writer then writes the received tuples back to disk.

Conceptually, the processing job is interesting because the tasks OCR Task, PDF Creator, and Inverted Index Task suggest having different computational complexities. In order to achieve a busy processing pipeline together with an economic job execution on the cloud, each task's degree of parallelization must be carefully balanced with respect to the other task.

As described in Section 2.1, many IaaS providers offer their VMs with ephemeral storage, i.e., it is not possible to store data inside the VM beyond its termination. Therefore, it is assumed that the set of input images is stored on a persistent storage service similar to Amazon EBS [12]. Right before the start of the processing job the storage service is mounted inside one particular VM. This VM then executes the tasks File Reader, PDF Writer, and Inverted Index Writer, so the input and output data is directly read or written from/to the storage service. The remaining tasks are executed on other VMs inside the cloud. Our overall goal is to find the largest possible scale-out of the job with high resource utilization before the VM serving the input data becomes an insuperable I/O bottleneck.

The evaluation experiments were conducted on our local Eucalyptus-based IaaS cloud testbed. Each subtask of the tasks OCR Task, PDF Creator, and Inverted Index Task was executed on a separate VM with one CPU core, two GB of main memory and 60 GB disk space. Amazon EC2 offered compute nodes with comparable characteristics (except for the disk space) at a price of approximately 0.10 $ per hour (as of September 2009). More details on the experimental setup can be found in the appendix.

The input data set for the sample job consisted of 4000 bitmap files. Each bitmap file contained a regular page of single-column text and had a size of approximately 10 MB. As a result, the overall size of the input data set was 40 GB. The PDF documents were created using the iText library [75]. In order to mimic the persistent storage service, we set up a regular NFS server.

4.4.2. Results

The results of the evaluation are depicted in Figure 4.3. The figure shows selected runs of our sample job with different degrees of parallelization. For each of those runs the average CPU utilization of all compute nodes involved in the execution over the time is

depicted. The CPU utilization was captured by successively querying the `/proc/stat` interface on each node and then sending the obtained values to Nephele's Job Manager to compute the global average for the respective point in time. Besides the CPU utilization chart, the figure illustrates the respective CPU or I/O bottlenecks which have been reported by our bottleneck detection algorithm in the course of the processing. Boxes with shaded areas refer to CPU bottlenecks while boxes with solid areas refer to I/O bottlenecks at the outgoing communication channels.

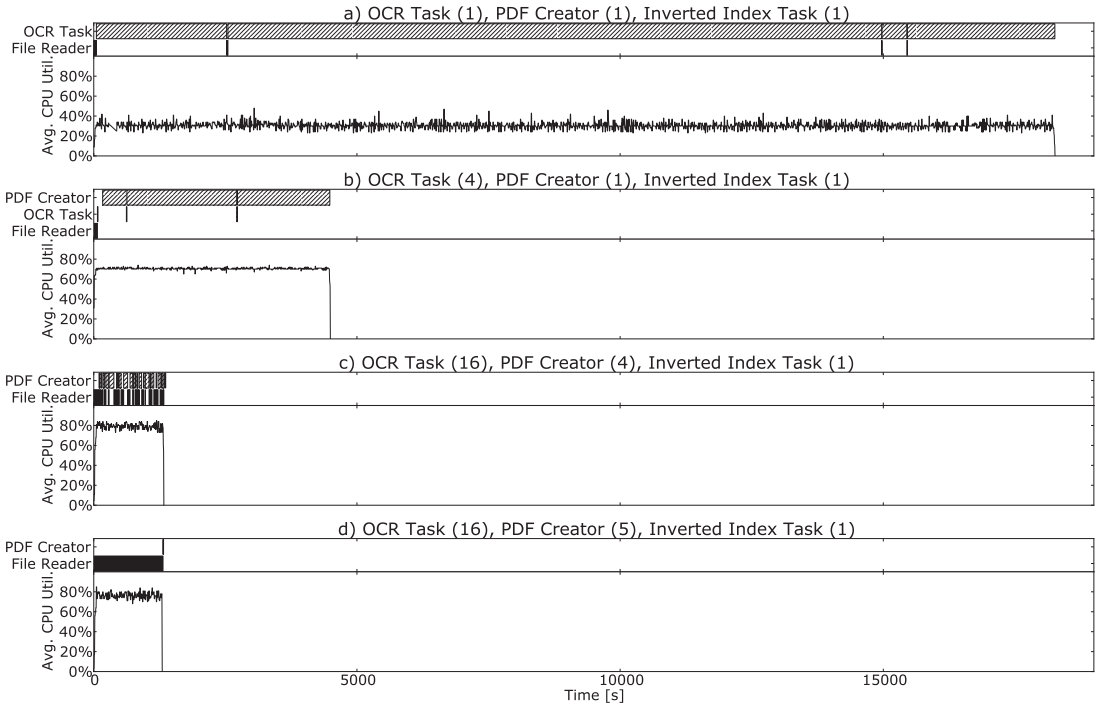


Figure 4.3.: Average CPU utilization and detected bottlenecks (shaded areas are CPU bottlenecks, solid areas are I/O bottlenecks) for different scale-outs.

Figure 4.3 a) depicts the first of our evaluation runs. As a first approach we used a parallelization level of one for all the six tasks of the sample job. As a result, the job execution comprised four VMs with the File Reader, the PDF Writer and the Inverted Index Writer running on one VM and the OCR Task, the PDF Creator and the Inverted Index Task each running on a separate machine.

The processing time of the job was about five hours and 10 minutes. In the entire processing period the average CPU utilization ranged between 30% and 40%. The reason for this poor resource utilization becomes apparent when looking at the bottleneck chart for the run. Almost the entire time the OCR Task was identified as a CPU bottleneck

by the bottleneck detection algorithm.

As a response to the observation of the first run, we followed the strategy to first improve the average CPU utilization by balancing the individual tasks' degree of parallelization according to their relative complexity. After having determined a reasonable parallelization ratio among the tasks we began to scale out. This approach requires that the computational characteristics of a task are independent of its level of parallelization (cf. Assumption 7).

We continuously increased the degree of parallelization for the OCR task and reexecuted the job. The average CPU utilization continued to improve up to the point where the OCR task had four subtasks (see Figure 4.3 b)). Up to a level of three the OCR task remained the permanent CPU bottleneck. However, at a parallelization level of four, the PDF Creator task became the dominant bottleneck. In this configuration, with seven VMs, the overall processing time decreased to one hour and 15 minutes while the average CPU utilization climbed up to approximately 70% throughout the entire execution time. Note that we did not have to wait for the intermediate runs to complete in order to deduce the final parallelization ratio between the OCR and the PDF Creator task. Since we knew the computational characteristics of both tasks would not change during the processing time, it was sufficient to observe only several seconds of each run and then to proceed to the next level of parallelization. For jobs consisting of several distinct processing phases, which interrupt the processing pipeline, a longer observation might be necessary.

After having done the initial balancing, we began to scale out both the OCR and the PDF Creator task at a ratio of four to one. In a configuration with 16 subtasks of the OCR Task and four subtasks of the PDF Creator task (see Figure 4.3 c)) we again encountered a change in the bottleneck situation. We witnessed frequent changes between the PDF Creator task as a CPU bottleneck and the communication channels of the File Reader task as an I/O bottleneck. The changes were caused by Nephele's internal buffer strategy for network channels. In order to achieve a reasonable TCP throughput, data is shipped in blocks of at least 16 KB size. The text patterns recognized by the subtasks of the OCR Task had an average size of about four KB, so the text pattern sometimes arrived at the subtasks of the PDF Creator task in batch and caused a temporary CPU bottleneck.

However, despite the frequent changes, the bottleneck bars in the diagram also indicate that the communication edge between the File Reader task and the OCR Task essentially became an I/O bottleneck which renders further parallelization of successive tasks unnecessary. This is confirmed by our final run depicted in Figure 4.3 d). After having added another subtask to the PDF Creator task we observed the communication channel between the File Reader and OCR Task to be a permanent I/O bottleneck.

Interestingly, the Inverted Index Task had no significant effect on the job execution. In comparison to the OCR Task and the PDF Creator task its computational complexity turned out to be too low. Moreover, the 4000 documents we used to populate the index only accounted for a memory consumption of a few MB. The channel congestion which may have occurred when transferring the index to the Inverted Index Writer task was too short to be detected by our system.

In sum, I think the evaluation provides a good example of the usefulness of our bottleneck detection approach. The initial job execution without parallelization (Figure 4.3 a)) took over five hours on four VMs to complete. Assuming an hourly cost of 0.10 $ per machine, this amounts to a processing cost of 2.40 $. Through the assistance of our bottleneck detection algorithm we could follow specific indications to scale out our sample job according to the complexity of each individual task. Although the final evaluation run (Figure 4.3 d)) spanned 23 VMs, the job already finished after approximately 24 minutes. This marks a comparable processing cost, however, at considerably savings in processing time.

4.5. Related Work

The performance analysis of distributed systems has been a field of vivid research in recent years. In general, the existing approaches can be subdivided into three different classes, based on their level of abstraction:

Performance analysis schemes on the lowest level of abstraction typically use code instrumentation or message interception to learn about the characteristics of a distributed application. Examples of those approaches are VampirTrace [84], TAU [117], or KOJAK [143]. While these tools generally provide very detailed information about the performance of an application, the information is often hard to translate into helpful insights about bottlenecks due to the sheer amounts of data produced.

On the middle level of abstraction, performance analysis tools no longer instrument the raw program code, but build upon a particular programming paradigm, such as workflow or a master/worker pattern. The framework for the respective programming paradigm can then define measurement points from which metrics can be derived. These metrics hint to specific performance issues in the user code or the degree of parallelism [88, 29].

The highest level of abstraction does not require knowledge of a specific programming paradigm, but rather considers the parallel application as a whole with a generic performance indicator. Examples of these performance indicators are the response time or other service level objectives of an n-tier application [80, 91, 90].

As a representative of the middle level of abstraction, Li and Malony [88] described a performance tuning that operates in several phases. The parallel program to be tuned is first instrumented and profiled by their TAU profiler. The recorded events are then aligned according to an abstract master/worker communication pattern. Based on this alignment, it becomes possible to create a performance model and various system metrics like the computation time or the derived efficiency of a worker. These metrics are then passed into a rule system that infers the causes for bad performance and presents them to the user, similar to the way our approach highlights the detected bottlenecks. Comparable work has also been presented by Cesar et al. [36]. The goal of their approach, however, is to determine a reasonable number of workers in a master/worker environment.

Several approaches studied performance aspects of applications following a pipelined workflow model over the past decades. Many of those considered more restricted workflow topologies such as linear chains instead of DAGs (such as [121]). In contrast to our work, the most common optimization objectives were throughput and/or latency given a fixed set of compute nodes. In [134] Vydyanathan et al. presented a heuristic based on estimations of processing and data transfer times. The goal of their approach was to schedule a DAG-shaped workflow on a fixed set of homogeneous processors in a latency-optimal manner while satisfying throughput requirements. As opposed to this work, our approach strives to maximize the system utilization on a variable set of computing resources.

In [29] Benoit et al. demonstrated a model for pipeline applications in grids. The pipeline model assumes a set of stages. Each of the stages comprises a data receiving, data processing, and data sending phase. Assuming a set of characteristics such as latencies and computing power, the model is capable of suggesting assignments of stages to processors. The model makes several limiting assumptions which make it considerably more restrictive than our approach, for example, each stage is required to process the same number of tasks. Moreover, it does not discuss the core question addressed in this paper, i.e., the detection of bottlenecks to infer reasonable degrees of parallelism.

As a representative of the highest level of abstraction, Chanda et al. discussed in [38] how to provide end-to-end profiles of transactions in multi-tier applications. They consider applications in which client requests are processed by a series of different stages. A stage may be a different process, a thread, an event-handler, or a stage worker thread. Through the algorithms and techniques introduced in their paper, the authors are able to track client requests through each of these stages and infer the amount of time the requests spent in them.

Apart from the field of distributed systems, bottleneck detection also plays an important role in other practical areas of computer science. For example, Kannan et al. [81]

presented an approach to detect performance bottlenecks in Multi-Processor System-on-a-Chip environments. Based on the idea of the dynamic critical path [131], their work aims at identifying components which contribute significantly to the end-to-end computation delay.

4.6. Summary

In this chapter I presented an approach to detect bottlenecks in parallel DAG-based data flow programs. The algorithm introduced as part of this approach is capable of detecting CPU as well as I/O bottlenecks and can therefore assist developers in finding reasonable scale-outs for their jobs.

Instead of directly relying on concrete system figures such as the throughput of the network interface, the new approach only considers these figures indirectly through a simple processing model. That way it is able to determine the bottlenecks solely based on the relationship among the tasks and the tasks' state information which it gathers within the data processing framework itself. With regard to the potentially spurious display of those system figures in VMs (cf. Section 2.3), this increases the robustness of our algorithms for parallel data processing on IaaS platforms.

Based on the parallel data processing framework Nephele, this chapter evaluated different strategies to obtain the tasks' state information which is required by our model at runtime. A first evaluation suggests that already a small number of iterations is sufficient to discover major performance bottlenecks and improve the levels of parallelization for the tasks involved in a processing job.

Guided by the feedback of the bottleneck detection algorithm, we manually adjusted the tasks' individual level of parallelism across multiple iterations of the same job in the scope of the presented evaluation. Technically, though, it is also conceivable to facilitate the scale-out automatically during a single execution of the respective job. In this case, however, the nature of the executed code must be carefully considered.

In the concrete case of the Nephele jobs, such automated scale-ins and scale-outs, i.e., the dynamic creation and deletion of subtasks at runtime, are not possible in general. Since Nephele allows users to include arbitrary code in their tasks, it cannot assume any knowledge about the tasks' internals. As a result, sudden changes in the number of a task's parallel instances at runtime might be incompatible with a task's or the overall job's internal logic. For example, incorporating a new subtask in a job which is already being executed might violate certain data distribution properties assumed by the preceding tasks in the processing chain on the one hand. On the other hand, destroying

a subtask is inherently unsafe as the subtask's internal state might be relevant to the processing result.

Solving this issue for arbitrary use code is a difficult subject. However, Chapter 7 discusses the usage of Nephele in combination with a higher layer programming abstraction. This programming abstraction features richer semantics and can provide hints which facilitate such automatic scaling strategies.

5. Mitigating I/O Variations with Adaptive Compression

Contents

5.1. Design Principles **72**
5.2. Adaptive Online Compression in IaaS Clouds **74**
5.2.1. Decision Model 74
5.2.2. Implementation in Nephele 76
5.3. Evaluation **79**
5.3.1. Adaptivity 79
5.3.2. Changing Data Compressibility 82
5.4. Related Work **83**
5.5. Summary **84**

As discussed in Chapter 2, many IaaS cloud providers run multiple VMs on the same physical hardware. Although this colocation of VMs is generally considered vital for the economic operation of IaaS clouds, the resultant sharing of the physical I/O infrastructure also makes these platforms susceptible to significant and unpredictable levels of performance degradation [82].

As a result of the current shortcoming in terms of VM isolation, a parallel data processing framework running on top of an IaaS cloud is constantly at the risk of experiencing I/O bottlenecks. Unlike the I/O bottlenecks I discussed in Chapter 4, these bottlenecks do not stem from inappropriate degrees of parallelization. They are induced by colocated VMs which, for example, have temporarily engaged in an I/O intensive operation.

A variety of projects is currently working towards improving the fairness of colocated VMs with regard to I/O performance [115, 146, 82]. However, since these proposals typically require modifications to the hypervisor, users of commercial clouds cannot benefit from those until their cloud providers consider them mature enough to be adopted.

For this reason we have devised an infrastructure agnostic approach to mitigate the effects of shared I/O in clouds which can improve the efficiency of parallel data processing without the assistance of the cloud providers, namely *adaptive online compression*.

The idea of adaptive online compression is to improve the I/O throughput by continuously choosing between different compression levels and applying them dynamically to the outgoing data stream. The compression level is selected by a decision model which constantly estimates the performance gain based on system metrics like the current CPU load, available I/O bandwidth, or the compressibility of the data.

In contrast to other approaches to cope with I/O bottlenecks in the context of parallel data processing, such as adapting the job's degree of parallelism or switching to other VM types, adaptive online compression is highlighted by its lightweightness. Changes to the compression level can be made with almost no overhead and take effect instantaneously. Moreover, compression does not have an influence on the structure of the processing jobs, for example the number of vertices in a Nephele DAG. This is advantageous because modifications to the structure of processing jobs at runtime generally require time-consuming coordinations between the affected worker nodes and the master node.

Although several adaptive online compression schemes have been introduced in recent years ([95, 77, 142, 87]), we found that these approaches are generally hard to apply in virtualized environments like IaaS clouds. The following section elaborates on their shortcomings and derives important design principles for adaptive online compression in the presence of hardware virtualization. Based on these principles, this chapter presents a novel adaptive compression approach and evaluates it through several experiments.

5.1. Design Principles

In recent years, a variety of adaptive online compression schemes has been presented ([95, 77, 142, 87]). While all these approaches have demonstrated their effectiveness in the domains they were originally designed for, we found them hard to apply in a virtualized setup like an IaaS cloud. Especially with regard to the characteristics of IaaS clouds, which have been discussed in Chapter 2, the following list discusses their deficiencies:

- **Dependency on offline training phase:** Several existing adaptive compression schemes (e.g. [142, 87]) require an offline training phase before they can be used. During that training phase, the decision model of the respective compression scheme is calibrated, so it can make reasonable compression decision when the system is in operation. With regard to IaaS clouds, these offline training phases suffer from two major problems: First, they must be performed in a verifiably unloaded system. In a commercial IaaS cloud, however, a VMs may spuriously appear to be idle, although the host system is heavily stressed by colocated VMs. Second, given that the performance characteristics of two VMs of the same type can differ

significantly in some cloud systems [114], the training phase must potentially be reexecuted after the instantiation of each machine. In sum, these training periods can account for considerable amounts of processing time and, thus, also increase the processing cost.

- **Dependency on accurate display of CPU utilization:** As illustrated in Section 2.3, the system metrics displayed inside a VM may be highly inaccurate. In particular, we discovered the display of the current CPU utilization to be significantly too low for many different virtualization techniques. However, the decision models of some adaptive compression schemes (such as [95]) rely on accurate display of these metrics. In practice, this can lead to unreasonable compression decisions in virtualized environments.

- **Small decision granularity:** Many existing adaptive compression schemes reconsider their compression decision on a granularity of several KB (for example [142, 87]). However, Section 2.3 highlighted caching effects which occurred for I/O operations in VMs in the scale of several MB or even GB. Due to these caching effects, the I/O throughput as observed in the VM temporarily appears to be overly high, but then, when the cache is periodically flushed, drops significantly. An adaptive compression scheme which evaluates the system throughput only based on several KB of data is therefore likely to make unreasonable compression decisions in the presence of hardware virtualization.

Based on these three shortcomings, the design principles for a new adaptive compression scheme which complies to the particular characteristics of IaaS platforms or virtualized environments in general can be formulated as follows:

- **No training phase:** The decision model of the new adaptive compression scheme must not require any offline calibration or training phase.

- **No decision based on CPU resources:** As the display of available CPU resources under high I/O load in VMs is likely to be skewed, the new decision model must not rely on it.

- **Embrace throughput fluctuations:** Unlike existing compression schemes, which try to adapt the compression level to the outgoing data stream on the granularity of KB, the new decision model shall focus on a granularity level of MB to allow for the possible throughput fluctuations this thesis highlighted in Section 2.3.

5.2. Adaptive Online Compression in IaaS Clouds

Based on the design principles elaborated in the previous section, I now present a new adaptive compression model which has been explicitly designed towards the particular characteristics of IaaS clouds [69]. Unlike existing approaches, the decision model of our adaptive compression scheme does not require a training phase. In addition, our mechanism takes the achievable application data rate, i.e., the data rate experienced by the application before compressing the data, as the foundation for the decision process. Although the application data rate at a particular compression level also involves aspects like CPU utilization, available I/O bandwidth or the compressibility of the data itself, it is only indirectly influenced by those. Therefore, our approach does not have to rely on the possibly inaccurate displays of those metrics inside the VM.

In the following I will explain the adaptive online compression scheme, notably its decision model, and describe the implementation in the parallel data processing framework Nephele. Although I have chosen Nephele as the technical foundation to show the effectiveness of our new scheme, its applicability is not limited to the domain of parallel data processing. Rather than that, the compression scheme can be beneficial for any kind of data transfer.

5.2.1. Decision Model

Similar to existing approaches our adaptive compression module is assumed to be placed between the application and the respective I/O layer. Instead of passing the data right to the I/O layer it is first intercepted by the adaptive compression module which, if considered beneficial, compresses the data according to a specific compression level. As already demonstrated by existing schemes (such as [87]), the entire adaptive compression/decompression logic can be encapsulated in a higher-level communication library and therefore becomes completely transparent to the application.

Following the idea of previous publications (e.g. [77, 142, 87]), our adaptive compression algorithm can choose between a fixed set of n compression levels. Each compression level thereby refers to a specific compression algorithm which is applied at the respective level. The individual compression levels must be ordered by their respective time/compression ratio. Compression level 0 stands for no compression. A variety of compression algorithms also offers parameters to influence their time/compression ratio. Therefore, it is conceivable to use the same compression algorithm at multiple levels but with different parameters.

Our adaptive compression scheme reconsiders the decision which compression level is to be applied every t seconds. Based on the amount of application data which has been

received from the application, (possibly) compressed, and passed to the I/O layer during that time span, we calculate the application data rate for these last t seconds. In case of network I/O the application data rate also includes the decompression time at the receiver because of the network's flow control mechanisms. The concrete decision algorithm to determine the compression level for the next t seconds is shown in Algorithm 4. The algorithm uses a series of auxiliary variables which are explained in Table 5.1.

Variable	Meaning
ccl	The compression level that is currently applied to the outgoing data stream. Initially, the variable is set to 0 (no compression).
ncl	The next compression level that shall be applied to the data based on the algorithm's decision.
c	A simple counter variable which stores how often the decision algorithm has been called since the last change of compression level. The counter is initialized with 0.
inc	A Boolean variable which indicates if the compression level has been increased in the scope of its last modification. Initially, the variable is set to $TRUE$.
bck	An array which stores the backoff values for the individual compression levels. Initially, all fields inside the array are set to 0.
cdr	The average application data rate which has been determined for the last t seconds using the compression level *ccl*.
pdr	The average application data rate which has been determined for the t seconds before the last t seconds. On the first call of the decision algorithm, *pdr* is set to *cdr*.

Table 5.1.: Explanation of the decision algorithm's variables.

As already mentioned, our decision model adapts the compression level in response to changes in the observed application data rate. This is also reflected in the structure of our algorithm, which distinguishes three major cases:

In the first case (lines 4-14) the application data rate during the last t seconds *cdr* (compressed with compression level *ccl*) does not differ from the data rate of the previous t second time span *pdr*. In order to cope with fluctuations in the data rate the parameter α is introduced. The parameter α defines in what range *cdr* may differ from *pdr* before our algorithm actually responds to the change. Small values of α allow our algorithm to detect the best compression level even if the performance gains between the respective compression algorithms are rather small. However, they also make the decision algorithm more prone to incorrect decisions because variations in the application data rate can also result from variations in the throughput of the underlying I/O system (such as the TCP

connection). During our experiments we found 0.2 to be a reasonable value for α.

Since our decision model cannot rely on any previous knowledge from an offline training phase, it optimistically switches to the next higher or lower compression level occasionally to see how the application data rate is affected. However, a fundamental aspect of our algorithm is that these switches occur less often for compression levels which have continuously led to improvements in the data rate. We achieve this behavior through an exponential backoff scheme (line 6). The decision to increase or decrease the compression level as part of such an optimistic switch depends on the variable *inc*. The variable *inc* indicates if the last change of compression level has been an increase or a decrease. Note that *inc* is usually updated outside of the displayed algorithm depending on the input parameter *ccl* and the return value *ncl*.

The second major case our algorithm has to handle is an improvement of the application data rate (lines 15-18). In this case our algorithm increments the backoff value of the current compression level $bck[ccl]$ by one. Thus, the algorithm will less often try out other compression levels from the current compression level given that no change in the data rate occurs.

The third and final case addresses a degradation of the application data rate (lines 19-27). In this case the algorithm reverts the last compression level change (lines 22-26). Moreover, it sets the backoff value for the compression level with which it has experienced the degradation ($bck[ccl]$) back to 0. Hence, optimistic switches to other compression levels again become more frequent for that compression level in the future.

Although our algorithm can make wrong decisions with respect to the chosen compression level, it can always react to degradations of the application data rate immediately (i.e., after t seconds) and revert the wrong decision. Good decisions are rewarded with increased backoff values. This ensures that any unnecessary probing of other compression levels decreases exponentially over time.

5.2.2. Implementation in Nephele

For our initial prototype we have integrated our adaptive compression scheme into Nephele's file and network channels. As illustrated in Figure 5.1, each output channel contains its own instance of the adaptive compression scheme. Consequently, Nephele can adapt the compression level for each outgoing data stream of a subtask individually. Moreover, the implementation is completely transparent to the subtasks, so there is no modification required to the encapsulated user code.

The compression decision is made after the serialization of the records which have been emitted from the user code. For performance reasons Nephele internally stores the

Algorithm 4 GetNextCompressionLevel(cdr, pdr, ccl)

```
1: d ← (cdr − pdr)
2: c ← c + 1
3: ncl ← ccl
4: if |d| ≤ α × pdr then
5:    {No change in application data rate}
6:    if c ≥ 2^bck[ccl] then
7:       {Backoff over, try another compression level}
8:       if inc = TRUE then
9:          ncl ← ncl + 1
10:      else
11:         ncl ← ncl − 1
12:      end if
13:      c ← 0
14:   end if
15: else if d > 0 then
16:   {Application data rate has improved}
17:   bck[ccl] ← bck[ccl] + 1
18:   c ← 0
19: else
20:   {Application data rate has decreased}
21:   bck[ccl] ← 0
22:   if inc = TRUE then
23:      ncl ← ncl − 1
24:   else
25:      ncl ← ncl + 1
26:   end if
27:   c ← 0
28: end if
29: return ncl
```

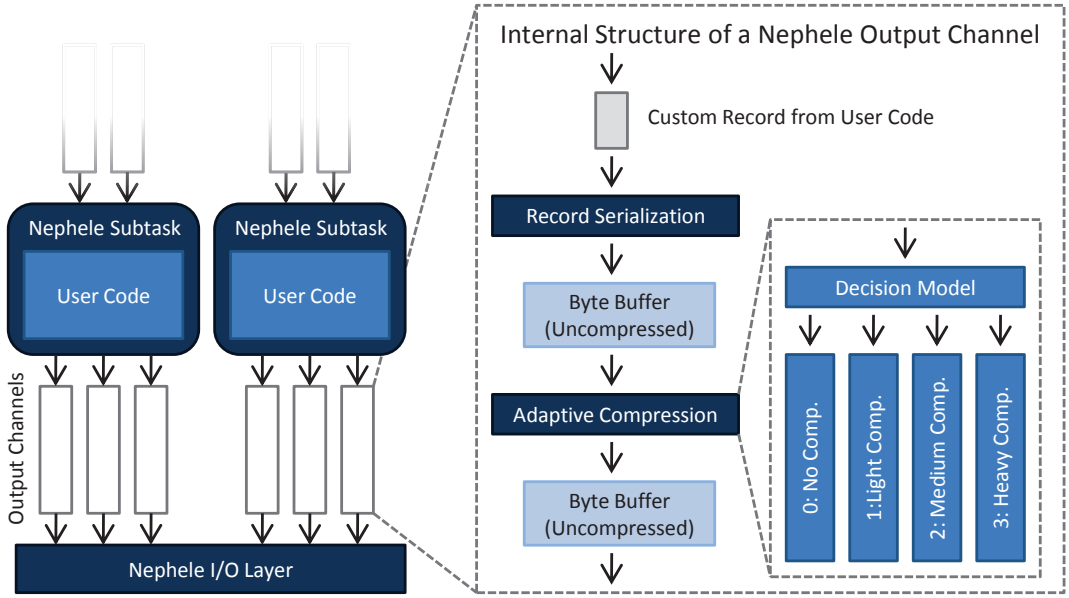


Figure 5.1.: Integration of the adaptive compression scheme in Nephele's communication channels.

serialized records in byte buffers of at most 128 KB size before passing them on to the I/O layer. With the new compression scheme enabled, the buffer is no longer handed directly to the I/O layer. Instead, it is passed to the adaptive compression component first. The compression component considers the size of the uncompressed data buffer as well as the period of time until the arrival of the next uncompressed buffer. Based on these two figures, the decision model is able to calculate the current application data rate. As described in the previous subsection, the decision algorithm is called every t seconds with this application data rate in order to assess the currently chosen compression level and to potentially adapt it.

The current implementation features four different compression levels. As in the description of the decision algorithm, compression level 0 again represents no data compression. Given the comparably high bandwidth of the available I/O interfaces, we have chosen the compression algorithms for the remaining three levels with regard to their compression speed. At compression level 1 (LIGHT) we use the QuickLZ compression library [110] which is highlighted by its fast compression speed. QuickLZ is also used for compression level 2 (MEDIUM) but with a setting which favors compression size over compression speed. For compression level 3 (HEAVY) we use the compression library LZMA [104]. Although LZMA is known to be significantly slower than QuickLZ, it generally offers a better compression ratio which might pay off if the available I/O bandwidth is low

enough.

The adaptive compression component writes the compressed data to a separate output byte buffer which is finally passed on to the I/O layer. After the compression process has been finished, the uncompressed data of the input byte buffer is discarded immediately. In order to simplify the decompression by the corresponding input channel of the receiving subtask, each output byte buffer is a self-contained unit of data. It contains all information required for the decompression, including meta information about the compression algorithm and the compression dictionary.

5.3. Evaluation

After having motivated and described our new adaptive compression scheme, I now want to present an evaluation of its performance based on a series of experiments. All of these experiments were conducted on our local Eucalyptus-based cloud using KVM-based VMs with paravirtualized I/O devices. The concrete setup of the VMs and the host systems corresponds to the one described in the appendix.

As a result of the tremendous caching effects for file I/O observed in Section 2.3, I will focus on network I/O only.

5.3.1. Adaptivity

The first series of experiments aims at demonstrating the ability of our adaptive compression approach to determine a suitable compression level for a given type of data and I/O bandwidth. We created a simple Nephele job which consists of two tasks (sender and receiver task) connected by a TCP network channel. The sender and the receiver task were concurrently executed on two distinct VMs. Each VM ran on a separate host system.

In order to evaluate the impact of different compressibilities on our approach, we conducted our experiments with three distinct files. The first two files were chosen from the Canterbury Corpus [21], a well-known compression benchmark. As a file with a high compressibility (HIGH) we chose the file `ptt5` from the benchmark which common compression libraries can compress down to 10-15% of its original size. As a representative of a file with moderate compressability (MODERATE) we chose the file `alice29.txt` from the corpus. Its compression ratio is about 30-50% depending on the algorithm used. Since the Canterbury Corpus does not offer files with a notably poor compressibility, we chose a standard JPG image of about 250 KB (refered to as `image.jpg` or LOW) as the third file for our experiments. Its compression ratio ranged between 90-95%.

	No concurrent TCP connection			One concurrent TCP connection		
	HIGH Mean (SD)	MODERATE Mean (SD)	LOW Mean (SD)	HIGH Mean (SD)	MODERATE Mean (SD)	LOW Mean (SD)
NO	569 (3)	**567** (7)	**566** (3)	908 (6)	896 (6)	**903** (6)
LIGHT	**252** (3)	629 (2)	688 (3)	**258** (3)	**624** (7)	927 (8)
MEDIUM	347 (6)	795 (5)	1095 (8)	367 (3)	840 (5)	1241 (42)
HEAVY	1881 (23)	5760 (25)	9011 (30)	1974 (24)	5979 (34)	9326 (30)
DYNAMIC	265 (4)	635 (4)	602 (3)	273 (3)	648 (16)	920 (13)

	Two concurrent TCP connections			Three concurrent TCP connections		
	HIGH Mean (SD)	MODERATE Mean (SD)	LOW Mean (SD)	HIGH Mean (SD)	MODERATE Mean (SD)	LOW Mean (SD)
NO	1393 (75)	1292 (67)	**1313** (39)	1642 (70)	1584 (120)	1638 (70)
LIGHT	**312** (14)	**756** (23)	1440 (87)	**358** (10)	1027 (65)	**1555** (17)
MEDIUM	378 (10)	896 (38)	1481 (27)	397 (3)	**953** (55)	1829 (100)
HEAVY	1985 (26)	6130 (31)	9597 (45)	1994 (21)	6218 (34)	9278 (49)
DYNAMIC	363 (22)	920 (18)	1452 (40)	411 (35)	1075 (37)	1865 (114)

Table 5.2.: Average completion times of the sample job using different statically chosen compression levels (NO, LIGHT, MEDIUM, HEAVY) as well as our adaptive approach (DYNAMIC). The completion times are subdivided by the compressibility of the data (HIGH, MODERATE, LOW) and the number of concurrent TCP connections.

In some of the experiments we colocated additional VMs on the same physical hosts in order to realistically assess the effects of shared I/O on our adaptive compression scheme. Each colocated VM on the sender's host system thereby established a separate TCP connection to another VM colocated on the receiver's host system and transmitted data as fast as possible.

In all the experiments the sender task repeatedly wrote the respective test files (either `ptt5`, `alice29.txt`, or `image.jpg`) to the network channel until a total data volume of 50 GB was generated and consumed by the receiver. During all the experiments t was set to two seconds and α to 0.2.

Table 5.2 summarizes the results of our experiments. The table shows the average completion times of the sample job for the different data compressibilities and the number of concurrent TCP connections which were established by the colocated VMs. The numbers in brackets denote the standard deviation. For comparison, the table also includes the average completion times when the compression level had been chosen statically before the execution and was not determined by our adaptive compression scheme at runtime. The numbers written in bold type mark the fastest execution.

As indicated in Table 5.2, the compression levels chosen by our adaptive compression scheme (DYNAMIC) led to average completion times which were at most 22% worse than the fastest average completion times with statically set compression levels. In many cases the completion times achieved with our adaptive scheme were between the first and second fastest completion times with statically set compression levels. Given that our algorithm has to perform some initial probing to determine the best compression levels, the results in these cases can be considered ideal.

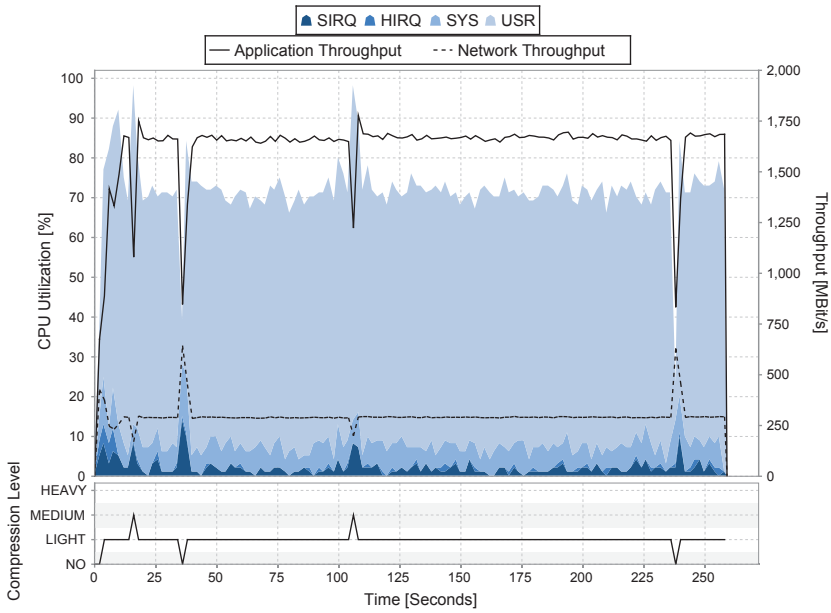


Figure 5.2.: Performance of our adaptive compression scheme with highly compressible data (HIGH) and no background traffic.

Figure 5.2 depicts the compression decisions our adaptive scheme made in such an ideal case, i.e., the highly compressible `ptt5` file (HIGH) with no concurrent TCP connections. The figure shows the sender's CPU utilization, application throughput, network throughput as well as the chosen compression levels over time. Due to the large differences in the compression/time ratios of the respective compression libraries, the decision algorithm can quickly determine the compression level LIGHT (QuickLZ, best compression speed) to result in the best overall application data rate. The figure also illustrates how the backoff mechanism we integrated in our decision algorithm reduces optimistic switches to other compression levels exponentially.

In case the performance differences between the respective compression levels are less distinctive, our decision algorithm may spuriously consider changes in the application data rate as fluctuations and continue the probing process. Figure 5.3 illustrates such

a case for the experiment with the poorly compressible `image.jpg` file (LOW) and two concurrent TCP connections. Lowering the value of α can help to counteract this behavior, however, it also increases the risk of wrong compression decisions due to regular fluctuations in the available TCP throughput.

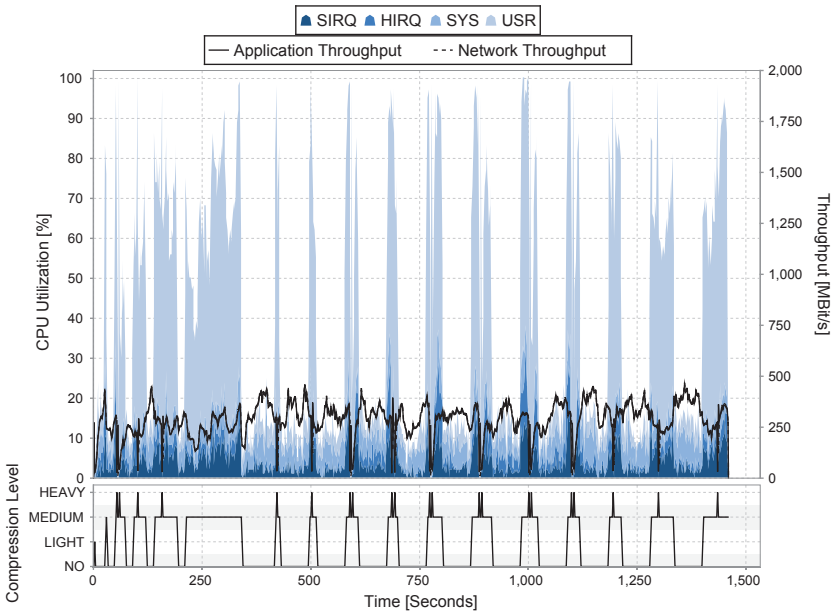


Figure 5.3.: Performance of our adaptive compression scheme with hardly compressible data (LOW) and two concurrent TCP connections.

5.3.2. Changing Data Compressibility

The second experiment examines how our adaptive compression scheme responds to severe changes in the data compressibility. Therefore, we reused the sample job from the previous adaptivity experiments and switched between the highly compressible file `ptt5` (HIGH) and the already compressed image file `image.jpg` (LOW) every 10 GB. Again, 50 GB of data were generated in total for this experiment. During the experiment, no background traffic was present.

The results of the experiment are depicted in Figure 5.4. Apart from some minor shortcomings our decision algorithm detected the changes in the data compressibility correctly and switched the compression level accordingly. Large backoff values for compression level 0 (no compression), which arose during the transmission of `image.jpg`, can lead to relatively late optimistic switches to a higher compression level. The reason for this behavior is the fact that without compression the application data rate is not affected

by the compressibility of the data. However, the opposite case is detected immediately by our algorithm.

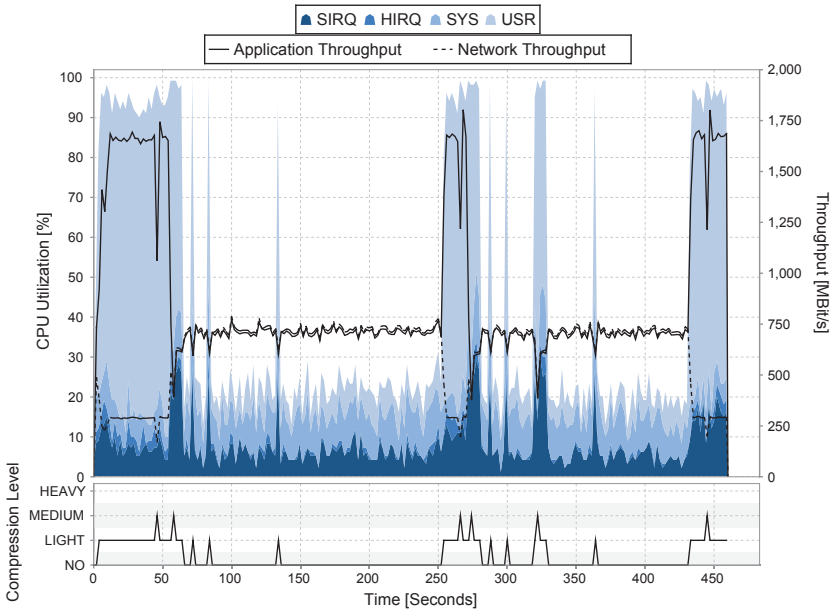


Figure 5.4.: Responsiveness to changes in data compressibility.

5.4. Related Work

In recent years, a variety of schemes for adaptive online compression has been presented. Although they aim at different fields of application, all these approaches build upon the same principle idea, namely dynamically selecting from a set of compression libraries to increase the overall data rate between two communicating parties.

With their network conscious text compression system (NCTCSys) [95] Motgi and Mukherjee focused on reducing the transmission times of HTML streams generated by a web server, e-mail text messages or large text files transmitted by FTP. Similar to our work, NCTCSys is capable of switching between different compression algorithms. However, the compression algorithm is chosen by directly evaluating a set of system metrics, including network bandwidth and the current server load. As pointed out in Section 5.1, this makes their approach prone to wrong compression decisions in virtualized environments like IaaS clouds.

Krintz and Sucu [87] presented a more general approach which is applicable to various kinds of input data. Like in NCTCSys, their decision model also includes CPU utiliza-

tion and network bandwidth as well as data obtained from an offline training phase. By exploiting a linear relationship between different algorithms' compression ratios, their decision model can quickly compare the estimated performance of the different compression algorithms used without testing them online. This enables their approach to avoid switching to unsuitable compression levels at runtime. However, it also makes their scheme dependent on the accuracy of parameters gained from the offline training.

Wiseman et al. [142] also implemented an adaptive compression system for various kinds of data. According to their approach, the compression algorithm is chosen based on each algorithm's compression time as well as the speed with which compressed blocks are processed by the receiver. A downside of their decision method is the usage of several hard-coded parameters, which need a short sampling phase with unloaded I/O and CPU in order to fit input data different from the ones used in their evaluation.

A system which does not rely on the direct measurement of system metrics was presented in [77] by Jeannot and Knutsson. Its main idea is to subdivide the compression and transmission process into a compression thread, a transmission thread, and a FIFO queue which connects both components. The decision to increase or to decrease the compression level depends on the current length of the FIFO queue. If the queue's length decreases (or increases, respectively), the compression level is lowered (or raised, respectively). A limitation of this system is the assumption that a higher compression level will lead to higher compression ratio, which is not always true, especially when the data is not compressible. Moreover, the system does not consider that applying stronger compression schemes might also increase the compression time.

5.5. Summary

In this chapter I presented an adaptive online compression scheme to mitigate the negative effects of shared I/O in IaaS clouds. With the help of this compression scheme, applications running inside the cloud can dynamically trade off CPU against I/O load in order to respond to sudden I/O bottlenecks and thereby enhance the overall application data rate.

Unlike existing adaptive compression approaches, the compression scheme presented in this chapter has been carefully designed in accordance with the system characteristics of today's IaaS platforms as examined in Section 2.3. In particular, the decision model of our compression approach does not depend on an offline training phase and accurate displays of the current CPU utilization. Instead, a feedback-based mechanism is utilized to calibrate the decision model during the actual data transfer. An exponential backoff scheme guarantees the convergence of the initial probing phase into a stable state given

that no changes in the application data rate occur. That way our adaptive compression scheme achieves a good balance between responsiveness and probing overhead.

Ultimately, the compression decision of our scheme is solely based on the data rate as experienced by the producer of the outgoing data stream. The decision model thereby averages the application data rate for several MB of data in order to alleviate the impact of fluctuating throughput rates as observed in Section 2.3. For network-based data transfers, when sender and receiver work simultaneously, the application data rate comprises all relevant factors for the compression decision, namely the compressibility of the data, the compression time required by the sender, the time for the actual data transfer, and the decompression time by the receiver.

For file-based data transfers, when the producer and consumer of the data are executed consecutively, our current decision model faces three problems. First, the application data rate experienced by the producer no longer reflects the effort for decompressing the data. However, given that for most compression libraries the computational complexity for data compressing is significantly higher than the one for decompression, this problem does not carry particular weight. Second, the adaptive compression approach can only be used to respond to I/O bottlenecks which occur while the producer is writing its data to disk. It does not help to mitigate I/O bottlenecks which the consumer encounters while reading the data. In fact, the compression decisions made by producer can even affect the consumer's performance adversely in this case. Finally, depending on the concrete virtualization technique used, the immense caching effects which have been highlighted for disk-based I/O in Section 2.3 may force the decision model to observe large quantity of outgoing data in order to compute a meaningful mean application data rate. This tremendously impacts the responsiveness of our approach and renders its applicability limited for mid-size data volumes.

Despite these limitations for file-based data transfers, I consider the new adaptive online compression scheme a valuable contribution to improve the efficiency of distributed applications. In extensive network experiments based on Nephele, the new adaptive scheme yielded job completion times which were at most 22% worse than the fastest completion times with statically set compression levels and improved the overall application throughput up to a factor of four. With respect to the mission statement of this thesis, the compression scheme therefore represents an important building block to counteract the current lack of performance guarantees of today's IaaS clouds and helps to improve the efficiency of parallal data processing on top of these platforms.

6. Topology Inference in IaaS Clouds

Contents

6.1. Analysis of Network Path Characteristics in Clouds 89
6.1.1. Inference based on Packet Loss 90
6.1.2. Inference based on Packet Delay 92
6.1.3. Discussion 94
6.2. Examining the Accuracy of Topology Inference 95
6.2.1. Obtaining Initial Similarity Values for the VMs 95
6.2.2. Accuracy of the Inferred Topologies 96
6.2.3. Transferring Binary Trees into General Trees 99
6.3. Implementation in Nephele 100
6.4. Evaluation 102
6.5. Related Work 103
6.6. Summary 105

Besides the I/O degradations their VMs may experience as a result of hardware virtualization or the colocation with other VMs, parallel data processing frameworks on top of IaaS platforms suffer from another shortcoming in contrast to their classic cluster setups, namely the lack of network topology information.

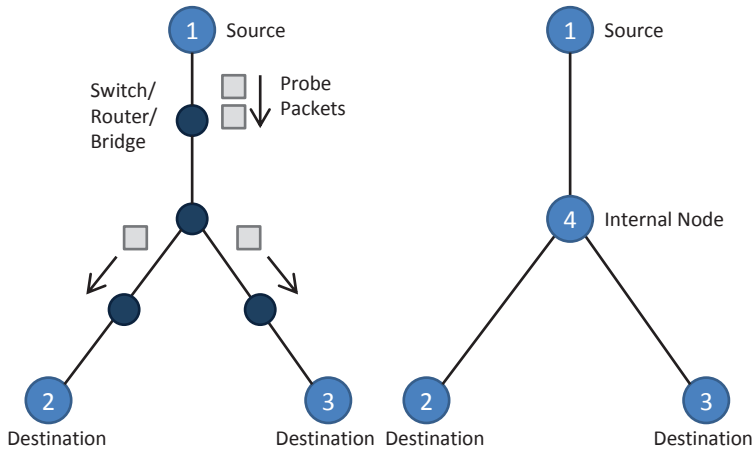
As explained in Section 2.3, many frameworks for parallel data processing offer to exploit knowledge about the network topology, i.e., the way the individual compute nodes are physically interconnected, in order to exploit data locality and reduce the risk of network bottlenecks during the execution of jobs [43]. In IaaS clouds, however, these details about the physical network topology are typically not exposed to a customer. Instead, the cloud customer has to rely on his VMs running "somewhere in the cloud" without knowing how many network components like network bridges, switches, or routers a data packet actually has to traverse in order to get from one of his VMs to another.

From a performance point of view, exploiting network topology information in the scope of parallel data processing is tempting. Considering the large volumes of data today's MTC-like processing jobs operate on, I/O bottlenecks in any of the above-mentioned network components can easily have a tremendous impact on the completion time and therefore the cost efficiency of the job. In addition to that, the available data throughput

within certain parts of the network topology might be significantly higher than in the rest of the topology. For example, as pointed out in Section 2.3, the communication between two colocated VMs can be over 50% faster than communication involving the actual physical network.

In order to make the underlying network topology of an IaaS cloud accessible to parallel data processing frameworks and allow them to use this knowledge for topology-aware scheduling, this chapter presents a scheme for topology inference in such setups [26, 25]. Following the general idea of network inference, the scheme's goal is to reconstruct likely topologies based on information about the individual network links that can be gathered at the application layer.

Although the scheme offers to take topology information gained from diagnosis tools like `traceroute` into account, it aims at also working in the presence of anonymous routers [148] (i.e., routers which do not respond to `traceroute` messages) and identifying network components underneath the network layer in the ISO/OSI stack [122], such as link layer switches and bridges. Therefore, this chapter puts strong emphasis on network topology inference based on end-to-end measurements (also sometimes called network tomography [130]).



(a) Physical routing tree (b) Logical routing tree

Figure 6.1.: The logical routing tree (b) as inferred from the physical routing (a) based on end-to-end measurements.

Figure 6.1 illustrates the overall idea. A set of network end nodes (for example VMs) is connected through a physical routing tree (Figure 6.1a). The physical routing tree's structure, in particular the internal nodes (e.g. network switches, routers, or bridges), is unknown to the end nodes. One or more source nodes from the set of end nodes

then send a series of probe packets to a set of destination nodes. Based on these probe packets, the end nodes can measure characteristics (like latency or loss) of the individual network path and correlate these to infer a likely logical routing tree (Figure 6.1b). The logical tree will not include all the internal nodes of the physical routing tree, but those at which the network path to two end nodes diverges.

Although there has been vivid research in the field of network topology inference recently (e.g. [41, 118, 98]), the applicability of this work in IaaS clouds has not been studied so far. Most previous approaches share a strong focus on large-scale networks, like the Internet, which are characterized by a large number of nodes, limited throughput, and considerable packet loss as well as latency. In contrast to that, the networks in today's cloud data centers are characterized by high throughput links and transfer latencies that are orders of magnitude smaller than the ones in wide-area networks. Moreover, the inference process in the presence of hardware virtualization has not yet been studied.

As a result, the chapter will start with an analysis of the different network path characteristics and carefully examine their suitability as proximity metrics for topology inference in virtualized environments. Based on this initial analysis, it will evaluate the accuracy to the inferred topologies and present a novel approach to improve the inference accuracy for typical data center network structures. Finally, a possible implementation of the scheme based on the Nephele framework is described and the impact of the newly acquired topology awareness is highlighted through a simple example job.

6.1. Analysis of Network Path Characteristics in Clouds

To infer likely network topologies based on end-to-end measurements, it is important to understand the characteristics of the paths which the probe packets travel. Bestavros et al. [30] established the theoretical foundation for topology inference in unicast networks. They showed that a broad class of link characteristics like throughput, packet loss rate, or packet delay can be used as a basis for a proximity metric given that these characteristics obey certain properties.

As a first step towards topology inference in clouds, this section will analyze the impact of different types of hardware virtualization on link *loss* and *delay*. Unlike in case of specialized network hardware, packets routed between VMs and their respective host system may experience unexpected delays or congestions due to high system load or scheduling strategies of the host's kernel. This initial analysis will highlight if the common assumptions [30, 98] for end-to-end-measurements in unicast networks still hold.

Although network throughput is also occasionally used for topology inference (for example in [98]), I deliberately skip this characteristic in the discussion for the following

reason: With respect to the fast interconnects in today's data centers, the nodes which are involved in the probing have to generate enormous amounts of traffic in order to expose any network bottleneck. Of course, this generated probing traffic has a negative impact on the performance of any network bound distributed application which is deployed on the nodes, too. In particular, I consider the high probing overhead to be incompatible with our strategies for dynamic resource allocation in the context of parallel data processing (cf. Chapter 3). One key aspect of these allocation strategies has been the integration of new VMs into a running processing job. However, in order to incorporate newly allocated VMs into the inferred network topology, the probing and inference process potentially has to be reexecuted several times during the job execution, when the network is already stressed by the parallel data processing itself.

To account for these data processing scenarios, this section studies the behavior of link loss and delay under different degrees of background traffic. All experiments presented in the following were conducted on our cloud testbed with 64 VMs hosted on eight physical servers. As in the previous experiments of this thesis, again the characteristics of the network links for KVM and XEN-based VMs are compared. For KVM we also considered VMs with unmodified device drivers (full virtualization) and modified ones (paravirtualization). A detailed description of the testbed can be found in the appendix. The confidence intervals, if shown in the plots, represent a confidence level of 95%. If not shown, confidence intervals have been small and omitted to improve legibility.

6.1.1. Inference based on Packet Loss

Packet loss is often considered as a link characteristic for topology inference [50, 51]. The idea is that a source node sends probing packets to at least two receiver nodes. The receivers measure their individual packet loss rate. From correlations in the loss rates it is then possible to deduce common subpaths between the source and the receivers.

Ideally, loss measurements are conducted in a multicast network so that a dropped packet affects the loss rate of all of its receivers in the same way. In unicast networks, the effect of a multicast transmission on the loss rate can be mimicked by a series of back-to-back unicast transmissions. However, a crucial prerequisite is that the loss rate of all unicast packets within a probe is positively correlated on a common subpath between the source and the receivers [98].

To verify whether this prerequisite is fulfilled in a virtualized environment, we let each VM of our cloud setup consecutively send probes consisting of two unicast packets to all other VMs within the same setup. The interval between two consecutive probes was 100 ms. Within one probe, the two unicast packets were sent back-to-back, i.e., with no intentional delay between the packets. Since both unicast packets are destined for

the same receiver (i.e., their common subpath equals the entire path both packets travel through the network) we expected to see either both packets of the probe arrive at the receiver or none at all.

Figure 6.2 depicts the loss rates we observed depending on the generated background traffic. Overall loss rate denotes the percentage of probes in which either one or both unicast packets did not reach the destination. 1-packet loss refers to the percentage of probes where only one of the two unicast packets arrived at the receiver. Details on the generation of the background traffic can be found in the appendix.

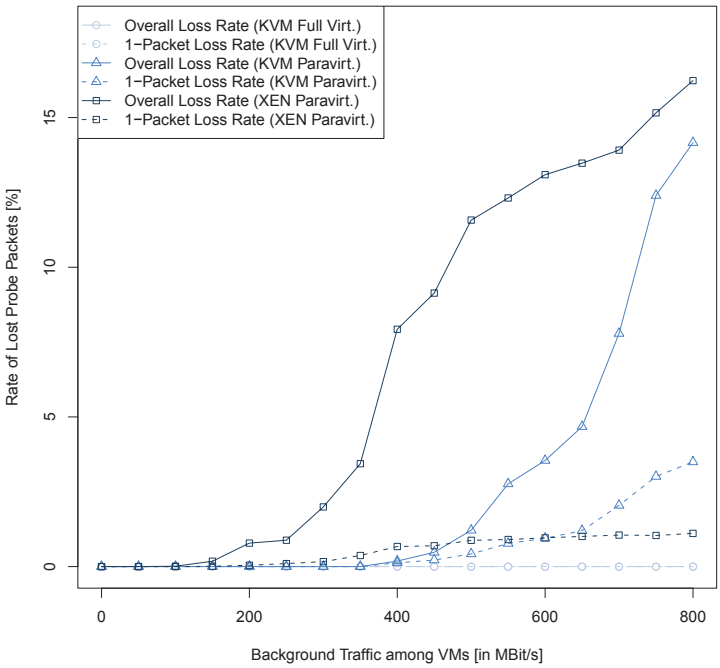


Figure 6.2.: Observed loss rates against background traffic for different types of virtualization.

As one fundamental obstacle we found that with full virtualization (KVM full virt.) we were unable to create any significant packet loss. At a background traffic of approximately 200 MBit/s per VM the overhead of the I/O virtualization became so high that the VMs fully utilized their assigned CPU core. Consequently, we hit a CPU bottleneck before we were able to cause any network overflow or congestion which may have resulted in packet loss.

The same experiment with paravirtualization showed different results. For both KVM and XEN-based virtualization we were able to observe packet loss of up to 15 or 16%, respectively. However, again it required considerable amounts of background traffic to be generated. Concerning the experiments with KVM and paravirtualization, it is also problematic that the correlation of packet losses within one probe is poor. Depending on the rate of background traffic, for 25% (800 MBit/s) to 65% (400 MBit/s) of all lost probes, one of the included unicast packets still reached its destination.

6.1.2. Inference based on Packet Delay

Topology inference based on packet delay follows a similar idea as the previous loss approach. Within one probe a pair of unicast packets is sent back-to-back to two receivers. As long as all unicast packets travel along the same subpath, they are expected to experience similar delays. Receiver pairs with a long shared subpath from the source are likely to have highly correlated delays while the delay of receiver pairs with a short common subpath is expected to diverge.

To analyze the impact of different virtualization techniques on the correlation of path delays, we let each VM of the cloud setup consecutively send probes consisting of two unicast packets to all other VMs. Again, the two unicast packets were sent back-to-back to the respective receiver. Like in the loss experiment, the common subpath of both unicast packets was identical to their overall path through the network. Hence, if packet delay on the common subpath was correlated, the second unicast packet would have to arrive at the receiver without any significant delay after the first one.

Figure 6.3 illustrates the average interarrival times between the first and the second unicast packet of a probe measured at the receiver. We distinguish between the interarrival times of those probes which have been exchanged between VMs on the same physical host (intra-host interarrival time) and those between VMs on different hosts (inter-host interarrival time). As a baseline, the plot also shows the packet interarrival time we measured for probes between the unvirtualized hosts.

The results indicate that paravirtualization increases the average interarrival time approximately by factor 10 compared to the unvirtualized baseline case. However, even with large amounts of background traffic the average temporal gap between the first and the second unicast packet's arrival is still considerably smaller than 0.1 milliseconds.

For full virtualization (KVM full virt.) the situation is different. Even at a relatively modest background traffic of 150 MBit/s per VM, the interarrival times grow to more than one millisecond. Given that the average packet RTT in today's local area networks is typically below one millisecond, it is unreasonable to assume any kind of correlation with respect to packet delay for this kind of virtualization.

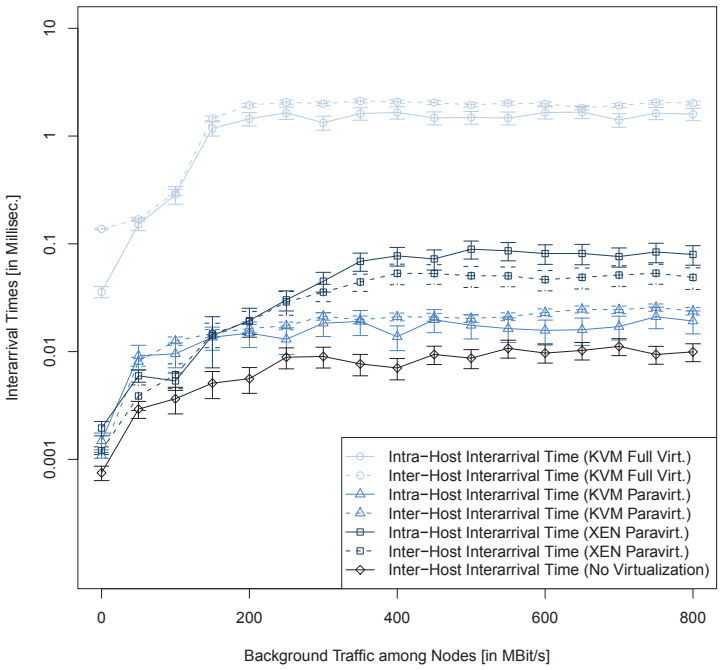


Figure 6.3.: Observed packet interarrival times against background traffic for different types of virtualization.

After having examined the impact of virtualization on the possible correlation of packet delay, we focused our second experiment on the effects of virtualization on the observable packet delay itself. Due to the lack of a global clock, measuring delay on a timescale of microseconds in a distributed system is a cumbersome task. As a remedy, we measured the packet RTTs instead. Figure 6.4 shows the results.

The results distinguish between RTTs which were measured between VMs running on the same physical host (intra-host RTT) and those running on different hosts (inter-host RTT). In general, the RTTs are highly influenced by the level of background traffic. With full virtualization (KVM full virt.), the average RTT rises up to approximately nine milliseconds at a background traffic of 200 MBit/s per VM. More importantly, the measured values show large variations for this type of virtualization, such that intra-host and inter-host RTTs appear essentially the same.

In contrast to that, the variance of the RTTs from the experiments with paravirtualization is much smaller. Moreover, we observed a distinct gap between the intra-host

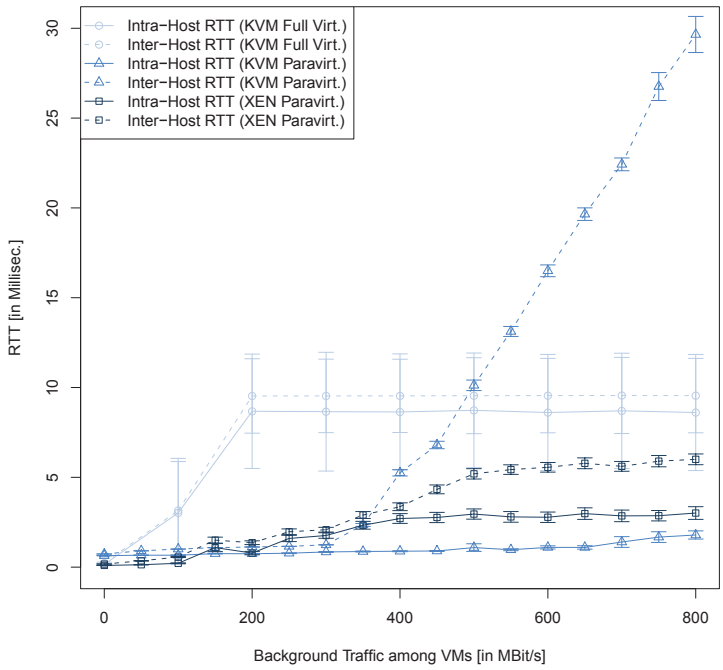


Figure 6.4.: Average RTTs against background traffic for different types of virtualization.

and inter-host RTT for both KVM as well as XEN-based VMs which continued to grow as the level of background traffic increased. Especially, for the KVM-based VMs the average inter-host RTT eventually rose up to almost 30 milliseconds.

6.1.3. Discussion

As an intermediate result of the effort to infer network topologies based on end-to-end measurements in IaaS clouds I can state that virtualization can have a significant impact on the observable characteristics of a network link.

For topology inference based on packet loss, both the KVM and the XEN virtualization layer destroyed the correlation of unicast packet loss on common subpaths, which is assumed by existing inference approaches [98]. In terms of packet delay, we experienced a similar problem for fully virtualized environments. Here the virtualization layer introduced significant gaps in the packet interarrival times which also render any assumption about correlated packet delays on common subpaths unreasonable.

Among all conducted experiments, observing link latency in paravirtualized clouds appears to be the most promising way to successfully deduce topology information among the involved VMs. Although we partly observed large increases in the RTTs under high load with both KVM and XEN, the individual values showed only little fluctuations at a particular level of background traffic. Moreover, we were able to measure statistically reliable differences between intra-host and inter-host RTTs.

6.2. Examining the Accuracy of Topology Inference

Having examined the impact of virtualization on the observable link characteristics loss and delay, the chapter will now deal with the actual topology inference process and its accuracy. As a reaction to our findings from the previous section I will focus on delay-based approaches and only consider paravirtualization.

Like most existing approaches (e.g. [109, 35, 50]) we attempted to reconstruct the logical routing tree through agglomerative hierarchical clustering. Starting with each VM as an individual cluster, this clustering technique progressively merges clusters based on a similarity metric γ until only one cluster is left. The approach requires an initial set of similarity values $\gamma_{i,j}$ for each pair of VMs i and j.

In the following I will discuss two different delay-based measurement approaches which can be used to construct $\gamma_{i,j}$ and contrast their performance in the inference process.

6.2.1. Obtaining Initial Similarity Values for the VMs

To obtain the required pairwise similarity metric $\gamma_{i,j}$ Coates et al. proposed a delay-based measurement technique called *sandwich probing* [41]. Compared to classic packet delay measurements, sandwich probing eliminates the need for synchronized clocks on the sender and the receiver node because it only measures delay differences.

As illustrated in Figure 6.5, a sender node s sends out a sequence of so-called sandwich probes. Each sandwich probe consists of three packets, two small packets destined for receiver j separated by a larger packet destined for receiver i. The second small packet is expected to queue behind the large one at every inner node of the routing tree (e.g. bridge, switch, etc.). This induces an additional delay Δd between the small packets on the shared links. Δd can be used as a similarity value $\gamma_{i,j}$ because the larger $\gamma_{i,j}$ becomes the longer the common subpath from s to i and j must be. The longer the common subpath from s is, the closer i and j must be in the logical routing tree. Since all sandwich probes originate from a single sender s, all inferred logical routing trees will have s as their root node.

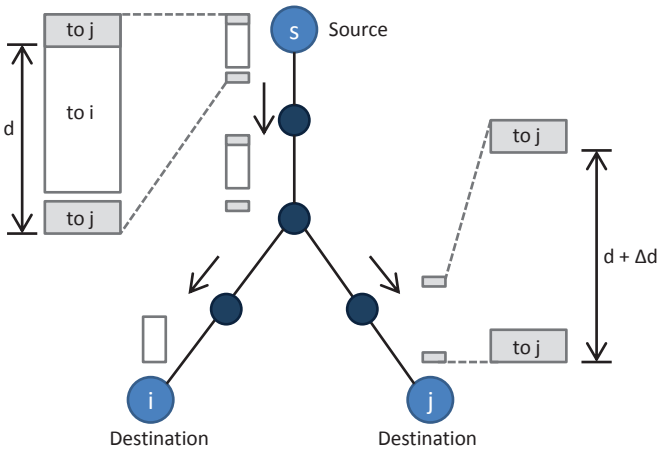


Figure 6.5.: The general idea of sandwich probing.

Another way to overcome the necessity for synchronized clocks is to measure path delay based on the *RTT* between all pairs of leaf nodes i and j. As opposed to sandwich probing the RTT measurements are not conducted from a single sender node s. Instead, each leaf node $l \in L$ (L is the set of leaf/end nodes) conducts its on measurements. However, the overall measurement complexity (i.e., the number of messages that must be transfered to obtain $\gamma_{i,j}$ for all pairs of $i, j \in L$) is still $\mathcal{O}(|L|^2)$. As a small RTT $r_{i,j}$ between two leaf nodes i and j indicates a close proximity in the inferred logical routing tree, the similarity value $\gamma_{i,j}$ must be defined as $\frac{1}{r_{i,j}}$.

6.2.2. Accuracy of the Inferred Topologies

To assess the accuracy of the inferred topologies that can be achieved based on end-to-end latency measurements in paravirtualized environments we collected samples for $\gamma_{i,j}$ using both sandwich as well as RTT probing. For the sandwich probing, we set the delay between the two small packets to $d = 10$ milliseconds. We also experimented with other values for d, however, found this one to provide the best overall results. In total, we collected approximately 50 probes for each pair of leaf nodes, each measurement technique, each level of background traffic, and each virtualization technology. For the actual clustering we used the agglomerative likelihood tree (ALT) algorithm as proposed by Castro et al. [35].

The accuracy of the inferred network topology is expressed as the distance between the inferred and the real topology in the Robinson-Foulds metric [112]. The Robinson-Foulds metric is based on an unrooted tree of nodes, each node of the tree must have a set of

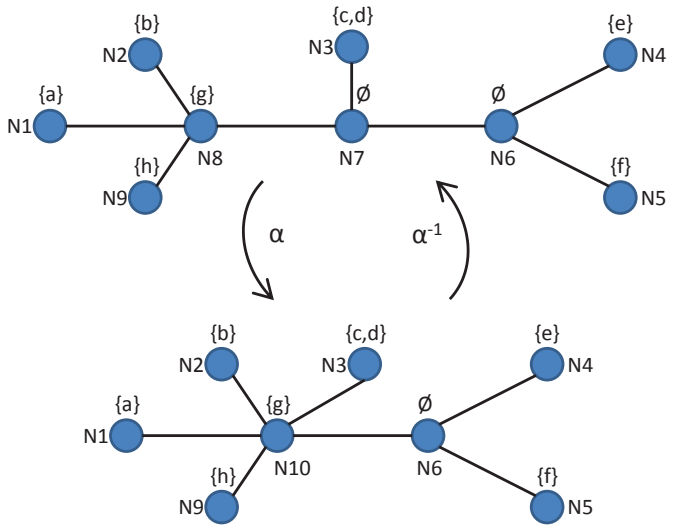


Figure 6.6.: The application of α and α^{-1} according to the Robinson-Foulds metric.

labels. For nodes with a degree greater than or equal to three, the label can be the empty set. All nodes with a smaller degree must have non-empty label sets. According to the metric, two trees T_i and T_j are considered to be the same if T_i and T_j are isomorphic, and the isomorphism also preserves the labeling [112].

Based on this definition, the Robinson-Foulds metric defines the elementary operation α, the contraction operation. The α operation can be applied to two adjacent nodes N_i, N_j of a tree. It shrinks those two nodes to a single node with the label of the new node being the union of the labels of N_i and N_j. Accordingly, the metric defines an inverse operation α^{-1} (the decontraction operation) which divides a node into two new nodes, the new nodes being joined by an edge. In order to construct the labels for those two new nodes, the original label set can be split arbitrarily as long the constrains mentioned above are preserved. Figure 6.6 illustrates the application of α and α^{-1} by means of an example tree.

The distance between two trees T_i and T_j is defined as $\frac{n_\alpha + n_{\alpha^{-1}}}{2}$ where n_α and $n_{\alpha^{-1}}$ are the minimum number of contraction and decontraction operations that have to be performed to transform T_i into T_j or vice-versa. To determine these numbers in the context of these experiments, we used the Phylogenetic Analysis Library (PAL) [47].

Figure 6.7 shows the accuracy of the inferred topologies for KVM as well as XEN-based VMs and sandwich as well as RTT-based probing against different levels of background traffic. Note that the inferred tree produced by the ALT algorithm is always binary [35]. Although there exist algorithms that can also infer network topologies based on general

trees, these algorithms also typically start by constructing a binary tree first and then apply stochastic methods to transform the binary tree into a likely general one [41, 51]. Hence, the constructed binary tree can be regarded as a robust starting point for the topology inference process. If an inference algorithm is unable to estimate a network topology accurately based on a binary tree, it will not be able to estimate it accurately based on a general tree either. An approach to transfer the binary tree into a general tree will be presented in the next subsection.

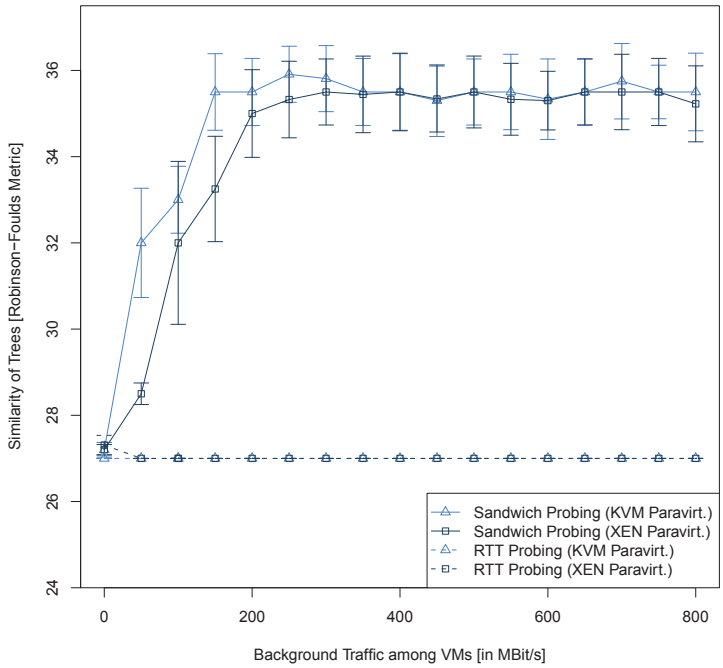


Figure 6.7.: Accuracy of the inferred topologies against background traffic using the ALT algorithm.

In the absence of background traffic the average distance of the inferred network topology tree to the real one centers around 27 for both virtualization and probing techniques. However, with increasing background traffic, the accuracy of those topologies that were inferred based on the sandwich probes starts to diminish. This can be explained by the increasing delay for intra-host VM communication under high background traffic.

As highlighted in Section 6.1 all packets which are passed to the physical network experience a significant additional delay even for moderate levels of background traffic. Although we found those additional delays to be relatively stable on a timescale of milli-

seconds, their variance is large enough to blur the subtle delay differences of intra-host VM communication. As a result, the delay differences for all receiver VMs which do not run on the sender VM's host suffer from a large variance and therefore impede the inference process in the presence of background traffic. In contrast to that, the topologies inferred based on the RTT probes, where each VM issued its own probe packets, are more stable towards increasing background traffic.

6.2.3. Transferring Binary Trees into General Trees

As pointed out in the previous subsection the topologies inferred by the ALT algorithm always follow the structure of a binary tree. Binary trees provide the largest number of degrees of freedom and thus are able to fit the measured data most closely [41]. Therefore they are a reasonable starting point to unbiasedly contrast the impact of different probing techniques on the inference accuracy.

However, the network topology in most data centers can be rather described by a general tree. The inferred binary tree can be considered an "overfitted" version of the physical network tree which includes more internal nodes than actually exist and therefore automatically decreases the inference accuracy in most network setups.

To overcome the problem of overfitting, several methods have been proposed (e.g. [41, 51]). Most of them apply computationally demanding heuristics to reconstruct likely general trees based on the initially inferred binary tree. Since most inference approaches were designed for large-scale networks like the Internet, which do not allow any assumptions about the structure of the routing tree, these heuristics are a reasonable choice. However, in contrast to the Internet, the network structure in data centers is much more regular. Typical network architectures of today's data centers consist of either two- or three-level trees of switches or routers [3]. Hosting multiple VMs on one server might add another level of depth to the tree, but, for example, a network topology tree with a depth of more than three in a single IP subnet is very unlikely to occur in practice.

The extension proposed in the following exploits this regularity. It is based on the assumptions that network topology trees with a depth greater than d are unlikely to occur. Moreover, it assumes that all leaf nodes are likely to have a similar depth in a data center network topology tree.

Our extension is subdivided into two operations: The first operation *reroot* takes the initial binary tree as input and chooses that inner node as the tree's new root node which minimizes the difference between the leaf nodes with the highest and the lowest depth. The operation accounts for the fact that the root node created by the clustering algorithm is not necessarily correct. For example, when using sandwich probing, the root node of the inferred binary tree is always the source of probe packets s. The rerooted

tree is then passed to the second operation *limitDepth*(d). This operation continuously identifies the leaf node with the highest depth, cuts it out, and appends it to its former parent's parent node as long as the depth of the tree is greater than d. Figure 6.8 illustrates the impact of our extension on the inference accuracy for different values of d. Compared to the initial binary tree (Figure 6.7) our extension reduces the Robinson-Foulds distance to the actual tree which represents our testbed's network topology by 16 to 21 (depending on d) on an average. To improve legibility the figure only shows the results for KVM (paravirt.) and omits confidence intervals.

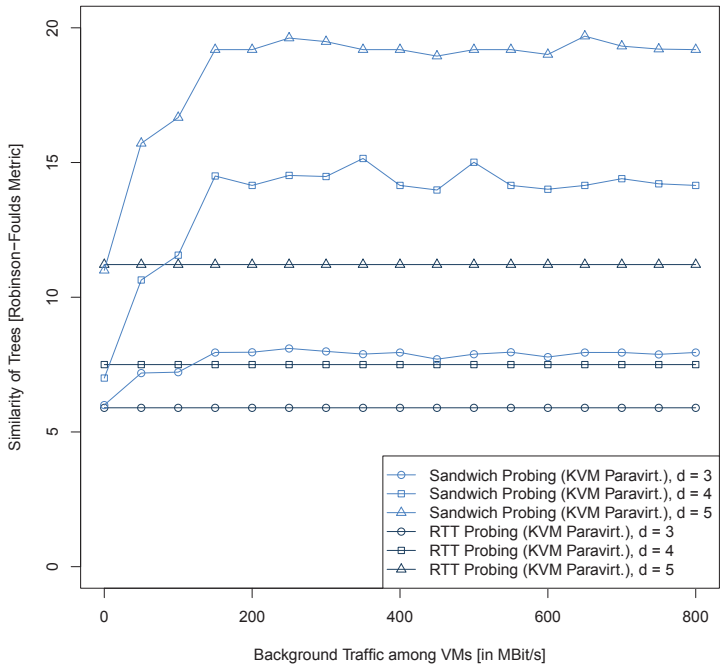


Figure 6.8.: Accuracy of the inferred topologies using the ALT algorithm and the depth limitation.

6.3. Implementation in Nephele

Following the previous discussion on the accuracy of topology inference based on end-to-end measurements in virtualized environments, this section now sketches a possible implementation of a concrete topology inference method as part of our parallel data

processing framework Nephele. Although we use Nephele as a technical foundation, the method has been devised as a lightweight background service which is applicable to all kinds of distributed applications in general.

Technically, our topology inference method is assumed to be deployed on a set of VMs which are connected through a tree-like but initially unknown network structure. Once the topology inference process has been activated, one VM from the set is elected as a master node. To simplify matters, in the context of Nephele, the master node is usually the VM which also runs Nephele's JobManager. After being elected, the master node starts sending out ICMP echo requests to all other VMs in the set. The goal of this operation is to obtain a first coarse-grained IP-level network topology.

As illustrated in Figure 6.9, the master node uses this IP-level network topology to subdivide the overall set of VMs into subsets of VMs which are in the same IP subnet. If (parts of) the IP-level network topology could not be obtained, for example because ICMP echoes are disabled by the cloud operator, a subset may also include VMs from different IP subnets. In the worst case, ICMP echo requests cannot be used to achieve any partitioning among the VMs, resulting in only one (sub)set containing all VMs.

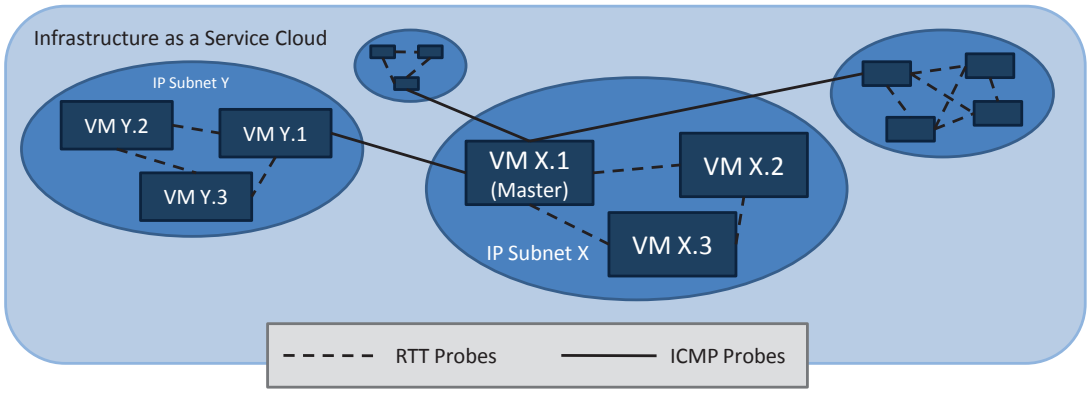


Figure 6.9.: Schematic overview of the topology inference method.

Then, in each subset, the end-to-end RTT measurements are started. The RTT probes are exchanges between all pairs of VMs in the respective subset. In contrast to the ICMP echo requests, the goal of these end-to-end measurements is to detect internal network components which operate underneath the IP layer. Examples of these components are link layer switches or bridges which connect the individual VMs to the physical network. The master node then collects then measurement data and executes an ALT-like clustering algorithm and the operation *reroot* and $limitDepth(d)$ on the data of each subset. Finally, the inferred network topologies of the individual subsets are integrated into the overall IP-level network topology.

Although our topology inference method does not depend on ICMP echo requests to be activated, the resultant partitioning of the VM set helps to reduce the overall number of RTT messages that have to be exchanged. By default, each VM conducts the RTT measurements every five seconds, whereas the data collection and clustering algorithm is triggered either every minute or immediately upon the arrival of a new VM. Consequently, the measurement and execution overhead is negligibly small for most cloud setups. For large-scale cloud setups (1000 or more VMs) the probing intervals can be increased in order to reduce the measurement overhead. Moreover, it is possible to delegate the execution of the clustering algorithm to a VM in each subset instead of doing the computation centrally on the master node.

To the distributed application which is supposed to take advantage of the inferred network topology, our method offers a well-defined interface. Through the interface the application can retrieve the overall inferred network topology. Moreover, the interface includes the possibility for callbacks to the application, so in case the inferred topology changes, for example due to newly allocated or terminated VMs or the possible migration of a VM, the application can be notified about that and respond to the changes.

6.4. Evaluation

We devised a sample job for Nephele in order to highlight the possible performance improvements that can be achieved by incorporating our new topology inference method.

The sample job consisted of eight data-parallel producer subtasks as well as eight data-parallel consumer subtasks. Each producer subtask was connected to a separate consumer subtask through one of Nephele's network channels. Moreover, each subtask was executed on a separate VM, so in total the experiment used 16 VMs hosted on two physical hosts of our local cloud testbed. All 16 VMs were in the same IP subnet.

We launched the sample job exactly one minute after having started Nephele's Task Managers on the respective VMs. As a result, the inferred network topology Nephele used for its scheduling was based on approximately 12 RTT samples per pair of VMs. After the execution of the initial clustering and the *reroot* operation, the *limitDepth*(d) operation was called with $d = 4$. During the execution of the sample job, each producer subtask continuously sent data to its connected consumer subtask. In total, all producer subtasks transmitted 200 GB of data to their respective consumers.

Figure 6.10 illustrates the possible impact of the new topology information on the job's performance and finishing time. When the network topology can be exploited for task scheduling (Figure 6.10a) Nephele can make sure that the respective pairs of consumer and producer subtasks are started on VMs which are colocated on the same physical

host. In this case the average network utilization increased to about 530 MBit/s on an average per pair of VMs and the processing job could be completed after approximately 400 seconds. Moreover, Figure 6.10a shows a high CPU utilization, which is also desirable in a cloud setup.

In contrast to that Figure 6.10b depicts the execution performance without a topology-aware scheduler. As in this case all available VMs appear equally suited to Nephele, the subtasks are assigned randomly to them. As a result of the subtask placement, about one half of consumer-producer pairs were sent to colocated VMs. The other half were forced to share their respective host's network interface to exchange data. This is also reflected in the average network throughput. Instead of 530 MBit/s, it dropped to about 210 MBit/s per pair of VMs. Thus, the job's completion time was also extended to approximately 16 minutes.

Finally, Figure 6.10c highlights the job performance which may result from a topology-agnostic scheduler in the worst case. Here all producer subtasks were assigned to VMs on one physical host, whereas the respective consumer subtasks were all scheduled to run on VMs on the other host. Since all producer subtasks essentially shared a single physical network device, in this case the average network throughput decreased to slightly over 100 MBit/s. Moreover, the average system utilization dropped below 10% and the completion time increased to over 30 minutes.

6.5. Related Work

Network topology inference has been subject to vivid research in recent years. In general, there exist two major classes of inference approaches: those which rely on the assistance of internal network nodes (such as routers) and those which do not.

The approaches falling into the first class (e.g. [22, 34, 78]) depend on receiving feedback messages from the internal network nodes. Most commonly, they use diagnosis tools like `traceroute` to discover the IP routers on the path to a destination host and afterwards try to reconstruct (parts of) the network's topology from the collected path information. While these network-assisted approaches provide accurate topology approximations in large, decentralized networks, they suffer from several shortcomings in data centers.

First, to the best of my knowledge, all network-assisted efforts to infer network topology rely on layer 3 of the ISO/OSI model (network layer). As a result, they are generally unable to detect link layer (layer 2) network components like switches or, more important in terms of virtualization, network bridges. Although there exist protocols like LLDP [72] to discover link layer (layer 2) components as well, these protocols mainly serve administrative purposes and cannot be used by arbitrary network participants.

Second, all network-assisted approaches fail to work if the network administrator of the cloud data center deactivates support for these types of diagnosis messages. This could be motivated by security considerations. At the network level this phenomenon is also referred to as anonymous routers [148].

The second major class of topology inference approaches, also known as network tomography [130], therefore solely relies on end-to-end measurements. The basic idea of network tomography is to measure particular characteristics of a network path during the exchange of probe packets among a set of network nodes. Typical characteristics observed are throughput, delay, or loss rate. When a sufficiently large number of measurements has been conducted, it becomes possible to calculate the correlation among the observed metrics and to infer a likely network topology [98].

Ratnasamy and McCanne [109] observed the shared loss rate at a set of receiver nodes in series of multicast packet transmissions from a single source. Based on the collected data, they proposed a clustering algorithm to infer the inner structure of a binary distribution tree. N.G. Duffield et al. later provided a formal proof for the correctness of the algorithm presented in [109] and extended it to general multicast topologies by introducing additional loss-based algorithms [50]. In [49] they further generalized their work to arbitrary estimable and monotonic performance metrics. Bestavros et al. [30] proposed a similar generalization with their MINT framework.

To overcome the poor availability of multicast in real-world networks, several projects also studied topology inference based on unicast end-to-end measurements. Coates et al. [41] presented a method to capture path delay in unicast routing tree topologies called sandwich probing. Instead of capturing the packet runtime at two distinct receivers, sandwich probing allows measuring delay differences at a single receiver and is therefore robust towards unsynchronized clocks. Based on this novel probing scheme, the authors devised a Markov Chain Monte Carlo (MCMC) procedure to infer the most likely network topology. In [35] Castro et al. demonstrated how to express the inference problem as a hierarchical clustering problem and proposed their ALT algorithm. Shih and Hero later extended their work by finite mixture models and MML model order penalties [118]. Ni et al. addressed the complexity of the proposed clustering algorithms and incorporated node joins and departures in the inference process [98]. Shirai et al. [119] as well as Tsang et al. [129] considered topology inference based on RTT measurements. However, none of the works examined the effects of hardware virtualization.

In the context of cloud computing, topology awareness has been considered by the following papers:

In [86] Kozuch et al. presented Tashi, a location aware cluster management system. Tashi features a so-called resource telemetry service which is capable of reporting the distance between a pair of VMs according to some user-defined metric. However, the

way the resource telemetry service obtains the location data of the VMs is not addressed in the paper.

Gutpa et al. [66] introduced a hosting framework for VMs. Their framework is able to deduce the traffic pattern of distributed applications running inside these VMs. Based on the observed traffic patterns, VMs are migrated inside the cluster such that the locality of a data transmission is improved.

Ristenpart et al. [111] discussed the location of VMs inside the Amazon EC2 cloud from a security point of view. Based on observations like common routing paths, common IP address prefixes, or packet RTTs, the authors examined the possibility to detect colocated VMs and exploit the colocation for attacks.

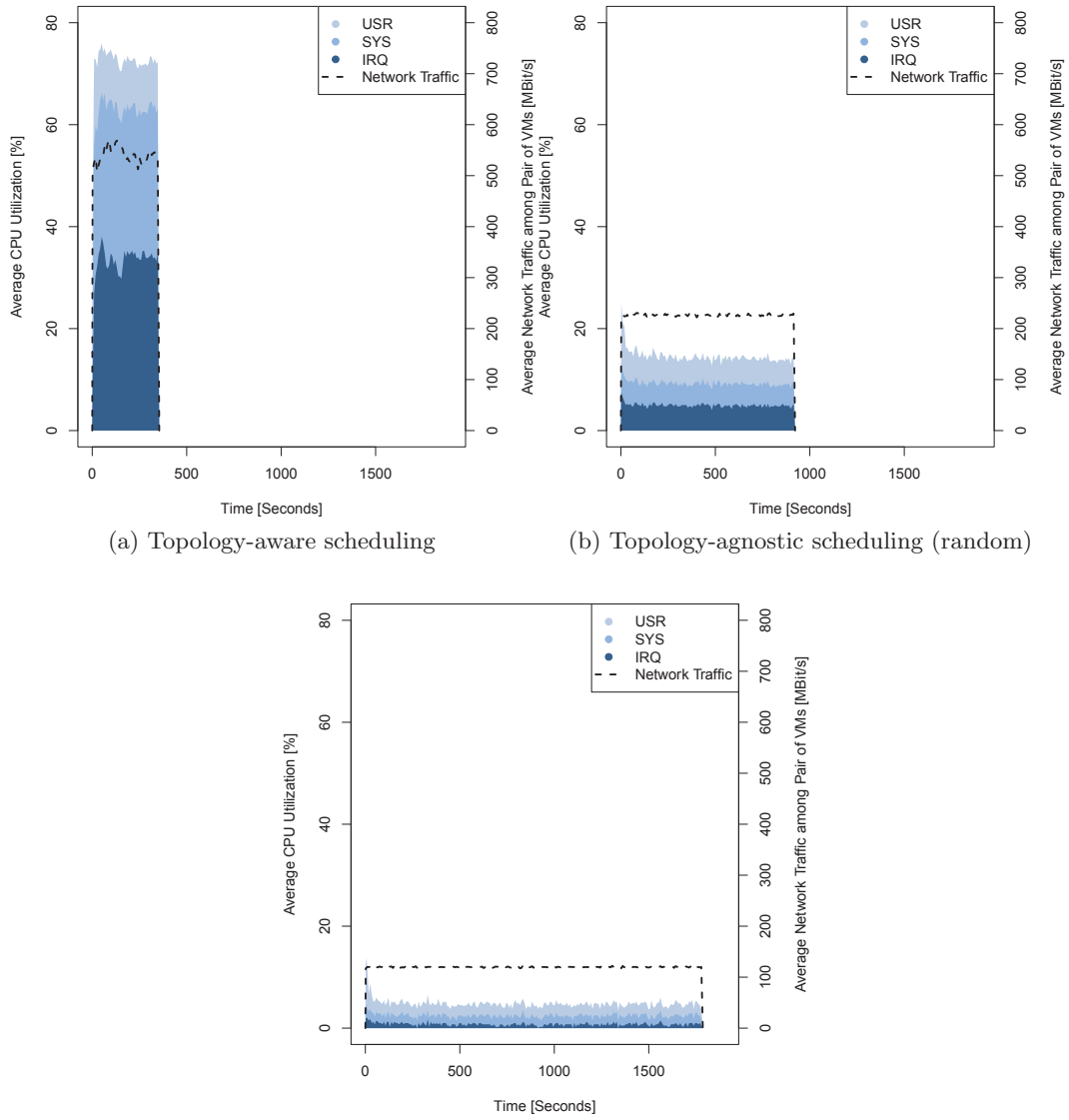
6.6. Summary

In this chapter I presented a lightweight scheme to reconstruct likely physical network topologies which connect a set of VMs in so-far opaque IaaS clouds. Thereby, the scheme takes a hybrid approach: It uses ICMP control messages to obtain a first coarse-grained IP-level network topology. Afterwards, the scheme performs end-to-end measurements to infer internal network nodes which operate underneath the IP layer.

As part of our inference efforts based on end-to-end measurements, the chapter provided an initial analysis of the network path characteristics loss and delay and examined the impact of hardware virtualization using the open source hypervisors KVM and XEN. Based on this analysis, it contrasted the accuracy of the inferred topologies for two different latency-based measurement approaches. Finally, the chapter proposed an extension to existing clustering-based inference algorithms. This extension allows to transform overfitted binary trees into general trees which are likely to describe the network topology of a cloud data center.

In sum, I can conclude that topology inference in IaaS clouds is a challenging but also rewarding subject. Even under moderate system load hardware virtualization has a significant effect on the measurable network path characteristics and destroys important correlation properties which are assumed by most inference approaches. Among all conducted experiments on our mid-size cloud testbed, RTT-based delay measurements led to the most accurate inference results with an average Robinson-Foulds distance to the actual topology of under six. However, we also observed large delays introduced by the virtualization layers (especially KVM) under high background traffic. In my opinion, the variance of these delays currently leaves little potential to reliably infer passive network components like link layer switches.

With respect to parallel data processing on IaaS platforms, however, the new scheme contributes to reduce the risk of I/O bottlenecks and fosters the exploitation of data locality anyway. Besides the reconstruction of the IP-level network topology, our scheme is able to reliably detect colocated VMs in paravirtualized setups. Since the network links between those colocated VMs are highlighted by an increased data throughput, it makes exploiting this kind of knowledge a favorable goal for topology-aware scheduling.



(a) Topology-aware scheduling

(b) Topology-agnostic scheduling (random)

(c) Topology-agnostic scheduling (worst case)

Figure 6.10.: System utilization and network throughput during the execution of the sample job using different scheduling strategies.

7. Interaction with Higher Layer Components

Contents

7.1. The Stratosphere Software Stack 109
7.2. The PACT Layer 112
7.2.1. The Structure of a PACT Program 113
7.2.2. The PACT Input Contracts 114
7.2.3. The PACT Output Contracts 117
7.2.4. Running PACT Programs on Nephele 118
7.3. Optimization Opportunities through the PACT Layer 120
7.4. Summary 122

The parallel data processing framework Nephele as well as the other contributions presented in the previous chapters are part of a larger software stack which is currently developed in the scope of the Stratosphere project [24, 5]. The Stratosphere project aims at facilitating complex and large-scale information management on top of IaaS platforms.

This chapter will present a brief overview of the Stratosphere software stack. In particular, it will highlight those layers of the software stack which are adjacent to the Nephele layer and discuss optimization opportunities resulting from the richer semantics of these layers.

7.1. The Stratosphere Software Stack

As illustrated in Figure 7.1, the Stratosphere software stack builds on an existing IaaS platform. The platform is expected to provide a large number of (typically virtualized) shared-nothing compute nodes.

On top of the IaaS platform there runs the so-called Stratosphere query processor. The Stratosphere query processor encompasses most of the software components developed in the scope of the Stratosphere project. In particular, it contains five distinct components:

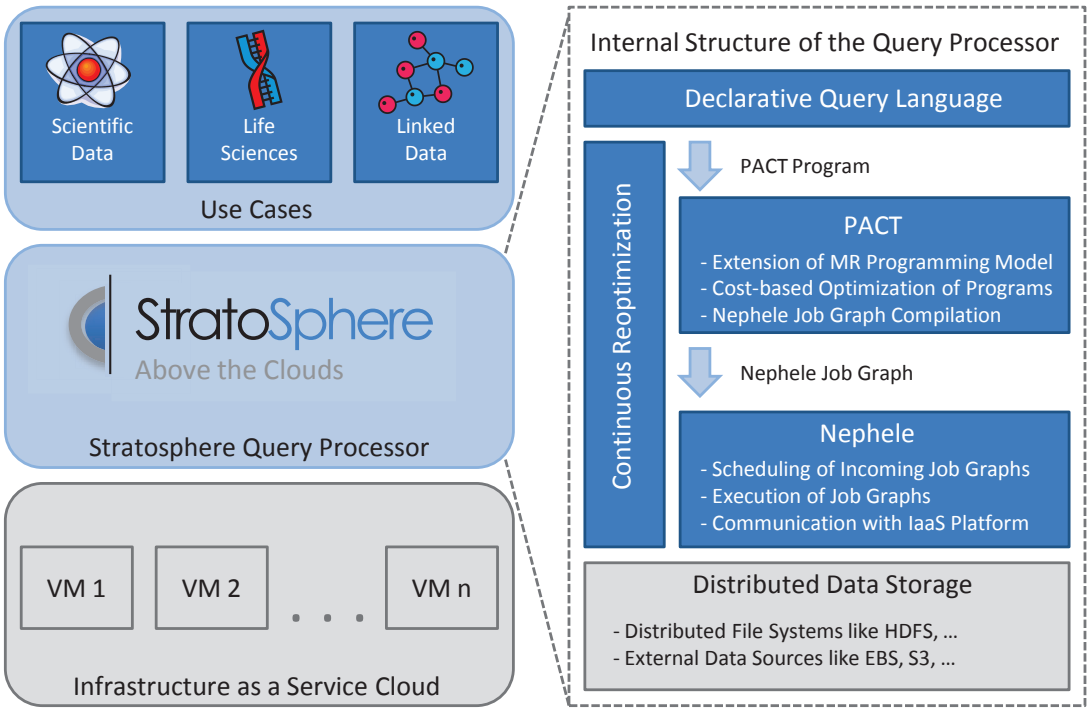


Figure 7.1.: Overview of the Stratosphere software stack.

On the lowest level of the Stratosphere query processor there is a distributed data storage layer. The main purpose of the data storage layer is to reliably store the large volumes of the queries' input and output data and to provide access to the data in a distributed fashion. Since storage does not take center stage on the Stratosphere research agenda, the layer currently utilizes existing solutions for distributed storage. This includes the distributed file system HDFS which is part of the Hadoop software stack [124] and stores the data directly on each VM of the cloud setup. However, the layer also contains bindings to external cloud storage service as described in Chapter 2.

On top of the distributed data storage layer the Nephele layer is located, the second component. As described in Chapter 3, Nephele's responsibility in the stack is the scheduling of incoming processing jobs, the so-called Job Graphs, as well as their efficient execution in the cloud environment. In particular, Nephele takes care of all resource management matters. It determines the number of VMs (and their types) which are necessary to run the respective Job Graph, assigns the job's subtasks to suitable compute nodes, and communicates with the IaaS cloud in order to allocate or release VMs.

The Nephele layer receives the incoming Job Graphs from the third component of the Stratosphere query processor, the so-called PACT layer. The PACT layer also lets its

users specify data processing jobs in the form of DAGs. However, it features a more abstract, higher-level programming interface in comparison to Nephele's Job Graph. Unlike Nephele's programming abstraction, the PACT programming model does not require the developer to fully implement the behavior of each of the job's tasks. Instead, it features a set of second-order functions, the so-called PACT input contracts, which can be considered a generalization and extension of the two well-known second-order functions map and reduce from the MapReduce programming model [6]. A developer simply has to attach his user code to the respective PACT input contract and can rely on the contract's guarantees in terms of the way data is passed to the user code at runtime. Particularly, the developer does not have to worry about the correct distribution of the data among different parallel instances of his user code during the job's execution.

While PACT input contracts provide certain guarantees on the property of the data that is passed to the user code, they do not specify how the system will achieve to fulfill these guarantees at runtime. In fact, for several PACT input contracts, there exist different strategies to fulfill the provided guarantees with different implications on the required effort for data reorganization. Choosing the best strategies for a submitted PACT program with regard to data transfer cost is therefore the main responsibility of the so-called PACT compiler, which is also part of the PACT layer. Based on cardinality estimates, the PACT compiler determines the cheapest execution strategies for the PACT program's respective input contracts. Then it adds the required functionality for the chosen strategies to the program and translates it into a Nephele Job Graph.

Compared to the Nephele programming abstraction, the PACT programming model is less flexible, but provides a more declarative way of writing MTC-like applications. Due to the specific set of input contracts a developer is bound to, the PACT layer can also make additional assumptions about the characteristics of the encapsulated user code. Since these assumptions generally do not hold for user code on the Nephele layer, exposing some of these characteristics to Nephele can yield some interesting opportunities for runtime optimizations, such as automated scale-in and scale-out. For this reason, the following sections will examine the interaction between Nephele and PACT during the job execution in more detail and point out sensible approaches for cross-layer optimization.

The fourth component of the Stratosphere query processor is the layer for continuous reoptimization. It is located orthogonally to the Nephele and PACT layer. The main purpose of the continuous reoptimization layer is to monitor the execution of a Nephele job and to take appropriate actions in case the execution exhibits an unexpected behavior in terms of the estimated processing time, data shipping cost, or related aspects. The continuous reoptimization layer can thereby take advantage of the different levels of abstraction the Nephele and the PACT layer offer. On the Nephele level, for example, the continuous reoptimization layer can build upon the bottleneck detection approach I presented in Chapter 4. However, it can also inject code into PACT programs to

detect execution problems which require a more precise understanding of the program semantics, like skewed data distributions.

The end user is envisioned to interact with the Stratosphere query processor by means of a declarative query language, the fifth component, which shares similarities with approaches like Jaql [31], DryadLINQ [150], or Hive [126]. In terms of its semantical richness, the declarative query language is more powerful than the PACT layer. Thus, queries are first translated into PACT programs to optimize their estimated physical execution cost and then passed on to the Nephele layer for the actual parallel execution. According to the current state of affairs, the declarative query language will provide a basic set of initial operators, but will also be extensible through custom operators and user-defined functions.

In its efforts to advance the state of the art in large-scale data processing on parallel, adaptive architectures, the Stratosphere project is targeted towards three real-world use cases: The first use case centers around the analysis of scientific data, for example from the area of climate research. In this domain, scientific simulations often produce data volumes in the scale of tera or even peta bytes [63] and call for massively parallel architectures to be analyzed in an efficient manner. The second use case deals with computational problems from the life sciences, especially information extraction and integration of unstructured data [89]. Finally, the third use case focuses on the analysis of linked data as it often occurs in social networks [76].

All three use cases are envisioned to make use of the declarative query language that is provided by the query processor. However, the use cases are also expected to develop a domain-specific set of operators and user-defined functions for the declarative query language with a wide range of different computational complexities.

7.2. The PACT Layer

As briefly sketched in the previous section, the PACT layer is situated on top of the Nephele layer within the Stratosphere query processor. More importantly, compared to the Nephele layer, it has additional knowledge about the behavior of the encapsulated user code. The combination of both aspects makes the PACT layer a favorable choice for cross-layer optimizations.

This section will describe the basic concepts of the PACT layer and explain how the PACT code is encapsulated in Nephele tasks at runtime.

7.2.1. The Structure of a PACT Program

Similar to Nephele jobs, the structure of a PACT program corresponds to a DAG. Each vertex of the DAG represents a particular task of the PACT program, whereas the DAG's edges model the data flow between those tasks. A user can also execute arbitrary sequential code as part of a PACT task, however, the way the user code of a PACT task is invoked is different compared to the Nephele layer. While the user code in a Nephele task is invoked exactly once during the job's execution, the user code wrapped inside a PACT task, or PACT for short, is typically invoked multiple times. Upon each invocation, the PACT user code is not provided with the entire input data from its predecessors in the DAG. Instead, the PACT layer separates the input data into disjoint subsets and calls the encapsulated PACT user code for each of these subsets individually.

The way these subsets of input data are composed for each PACT depends on the respective PACT's input contract. As the name implies, an input contract can be considered an agreement between the PACT layer and the PACT user code. It provides a specific guarantee on the properties of the input data that is passed to the PACT user code on each invocation. For each PACT, an input contract is mandatory, i.e., the user code on the PACT layer must be designed to fit the respective contract's interface.

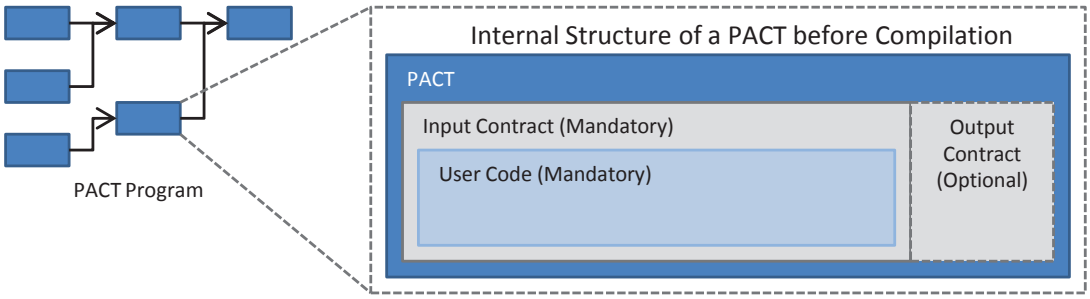


Figure 7.2.: The basic structure of a PACT program.

Besides the concepts of input contracts, the PACT programming model also includes so-called output contracts. Unlike input contracts, output contracts are optional. A developer can attach these output contracts to his particular PACTs in order to indicate that the output data of the encapsulated user code will obey certain properties. With regard to the programming abstraction, output contracts have no direct benefit to developers. However, they provide valuable information about the user code's behavior to the PACT compiler. The PACT compiler in turn can exploit this information to optimize the necessary data reorganization between the individual PACTs. In sum, the structure of a PACT program can be illustrated as in Figure 7.2.

7.2.2. The PACT Input Contracts

Currently, the PACT programming model features five distinct input contracts. Each input contract provides different guarantees on how a PACT's input data is split into subsets and passed on to the encapsulated user code.

In order to understand the concrete properties of these subsets, it is important to know that the PACT layer is based on a key-value data model. This means, unlike Nephele which does not understand the semantics of its records at all, the PACT layer can at least split each unit of data into a separate key and a value. The key and the value themselves again can be of a user-defined type and completely uninterpretable to the PACT layer. However, the key type must allow for the ordering of keys.

The Map Input Contract

The Map input contract corresponds to the well-known second-order map function from the MapReduce paradigm [43]. It can be considered to be the simplest of all currently available input contracts. As illustrated in Figure 7.3, the Map input contract states that the user code is invoked exactly once for each key-value pair of the input data set.

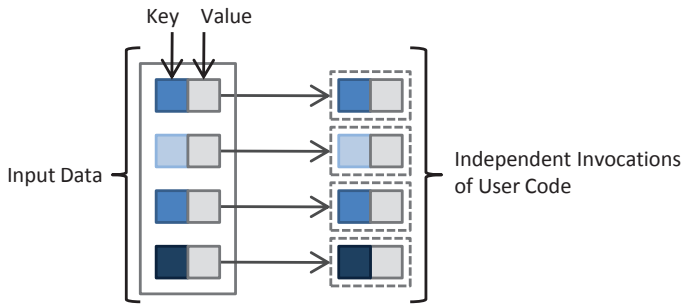


Figure 7.3.: Grouping of input data according to the PACT input contract Map.

The Reduce Input Contract

According to the the PACT input contract Reduce, all key-value pairs of a PACT's input data with an identical key are grouped. The user code which is attached to the Reduce contract is invoked for each of these groups independently (Figure 7.4). This also corresponds to the behavior of the second-order reduce function from the MapReduce paradigm.

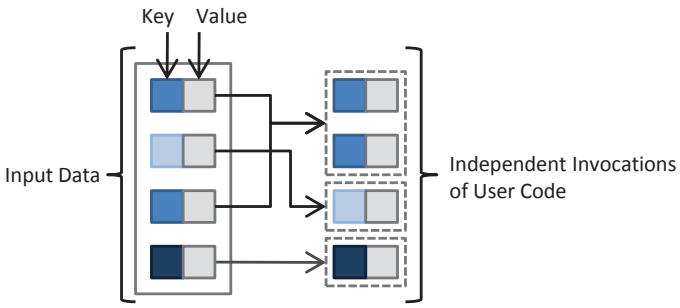


Figure 7.4.: Grouping of input data according to the PACT input contract Reduce.

The Cross Input Contract

Unlike the Map and Reduce input contract, the PACT layer also features so-called multi-input contracts. The user code attached to those multi-input contracts expects to receive data from two distinct data sources as input. Therefore, multi-input contracts also construct the subsets for the user code based on two different input sets.

One of these multi-input contracts is the Cross input contract. As illustrated in Figure 7.5, it builds the Cartesian product of the two inputs. All pairs in the Cartesian product are then processed independently by separate calls of the user code.

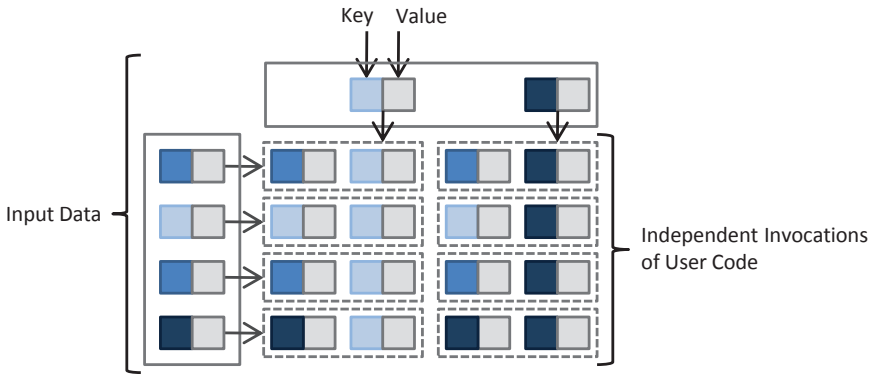


Figure 7.5.: Grouping of input data according to the PACT input contract Cross.

The Match Input Contract

As illustrated in Figure 7.6, the Match input contract is another multi-input contract. With this contract, all combinations of key-value pairs with identical keys are built over

the provided input data sets. Afterwards, all combinations are processed independently by separate invocations of the attached user code. From a database researcher's point of view, the Match input contract resembles an equi-join on the key.

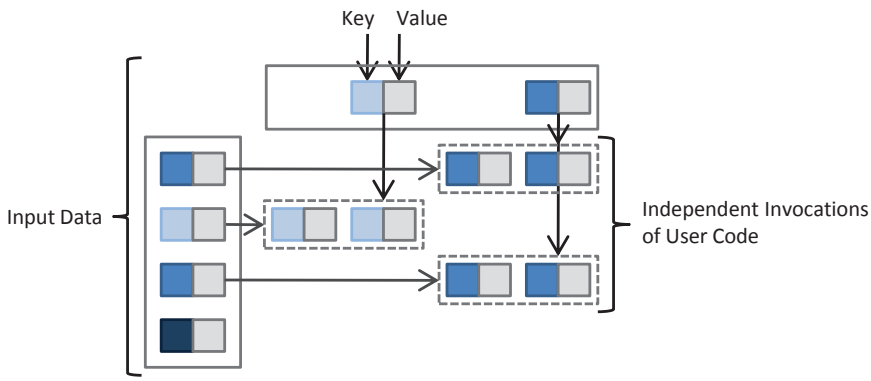


Figure 7.6.: Grouping of input data according to the PACT input contract Match.

The CoGroup Input Contract

Finally, the current version of the PACT programming model also features the so-called CoGroup input contract, which is also a multi-input contract. With the CoGroup contract, key-value pairs with identical keys are grouped for each input. Then, those groups of all inputs with identical keys are processed together by the user code (Figure 7.7).

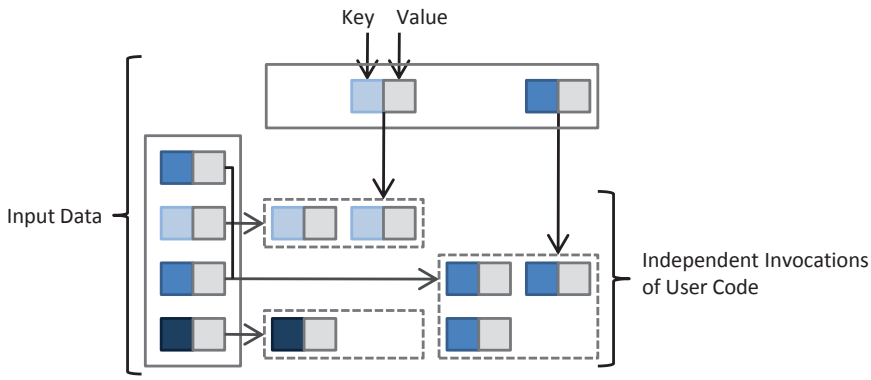
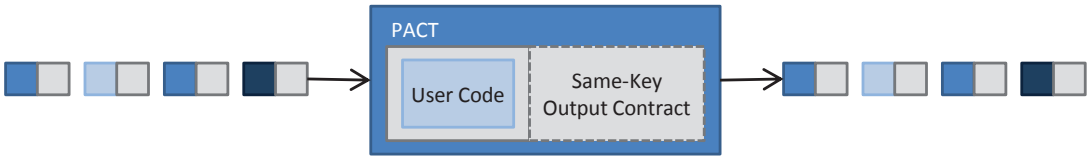


Figure 7.7.: Grouping of input data according to the PACT input contract CoGroup.

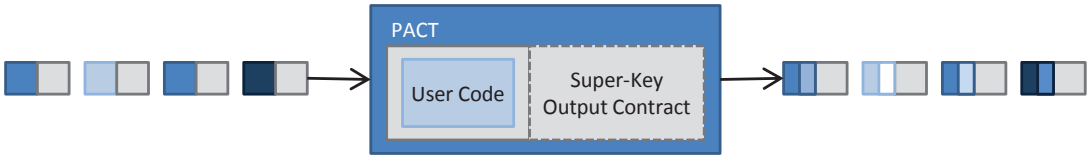
7.2.3. The PACT Output Contracts

While a PACT input contract can be regarded as a warranty from the system to the user code, PACT output contracts work the opposite way. By attaching an output contract to his user code, a developer provides a guarantee to the system that the output of the user code will obey certain properties. As mentioned previously in this section, output contracts are an optional component of a PACT task. Omitting an output contract will have no effect on the correctness of the computation. In the worst case, the lack of an output contract will lead to a less efficient job processing because the PACT compiler cannot apply certain optimizations. Spuriously adding an output contract to a piece of user code, however, can lead to wrong processing results since the system then assumes properties of the user code's output which are not actually true.

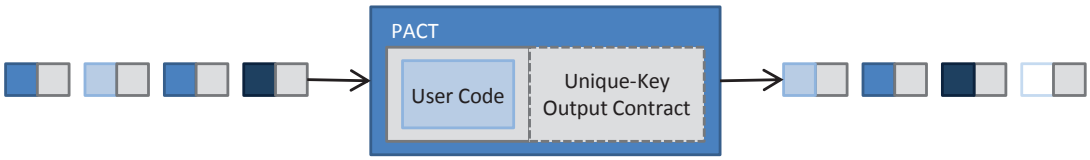
Currently, the PACT programming model features three different output contracts. Figure 7.8 summarizes them.



(a) Same-Key output contract



(b) Super-Key output contract



(c) Unique-Key output contract

Figure 7.8.: Overview of PACT output contracts.

With the Same-Key output contract (Figure 7.8a) the developer can assert that his user code does not alter the keys of the consumed key-value pairs. As a result, each key-value pair which is generated by the user code must have the same key as the key-value pair it has originally been generated from. For multi-input contracts like Cross, Match, and

CoGroup, the Same-Key output contract must also state to which of the input data sets the contract applies.

The Super-Key output contract (Figure 7.8b) allows a developer to express that each key-value pair which is generated by the user code must have a superkey of the key-value pair it has been originally created from. Similar to the Same-Key output contract, the Super-Key contract requires an additional annotation when used with multi-input contracts in order to clarify to which input data set the contract refers.

The Unique-Key output contract (Figure 7.8c) differs from the two previous output contracts as it does not specify a relation between those key-value pairs which are fed into the user code and those that come out. Instead, the Unique-Key output contract simply states that all key-value pairs that are generated across all invocation of the user code in the scope of the processing will have a unique key.

7.2.4. Running PACT Programs on Nephele

Once a user has finished writing his PACT program, he submits the program to the PACT compiler. Based on the provided contracts the PACT compiler then attempts to compile the program into a Nephele Job Graph which fulfills all of the user's input contracts on the one hand, but minimizes the necessary data transfer over the network when executed on multiple VMs on the other hand. As part of this transformation step the compiler also adds annotations concerning the concrete channel types to use between two connected Nephele tasks. Nephele in turn can apply these annotations to enforce the colocation of tasks on the same VM at runtime.

After being compiled, Nephele treats the PACT program like a regular Job Graph. As explained in Section 3.2.3, the received Job Graph is spanned into an Execution Graph and prepared for the parallel execution on the IaaS cloud. In the current implementation, the PACT program adds an annotation in terms of the desired degree of parallelization to the submitted job. Nephele then uses this annotation for the scale-out of each of the job's tasks.

At runtime, multiple parallel instances of each PACT task are executed on the IaaS cloud in a parallel and distributed fashion. Each of these parallel instances is encapsulated into a separate Nephele subtask which communicates with the respective Nephele TaskManager.

As described in Section 7.1, the PACT compiler can often choose from several different strategies how to fulfill a PACT input contract at runtime. Depending on the respective input contract, the compiler therefore may have to inject additional runtime code besides the original PACT user code into a Nephele task. The purpose of this PACT runtime

code is to preprocess the incoming or outgoing data according to the guarantees of the input contract. Figure 7.9 provides an overview of the the different layers of code which run inside a Nephele task during the job's execution.

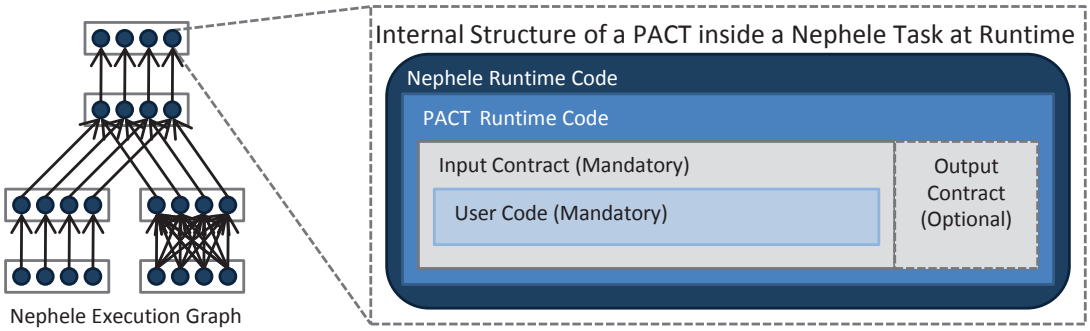


Figure 7.9.: Overview of different layers of user code which are executed inside a Nephele task at runtime.

On the lowest level, the Nephele task contains the Nephele runtime code. The Nephele runtime code provides the communication interfaces for the user code. Moreover, it is responsible for the allocation and release of communication buffers, the propagation of changes in a task's execution state, and possibly data compression. Since this code is part of the Nephele framework itself, Nephele is aware of its semantics. In particular, the framework can keep track of the encapsulated user code's communication behavior, i.e., at what points of the user code's execution incoming records are read from and outgoing records are written to the Nephele channels.

Inside the Nephele runtime code the PACT runtime code is executed. Contrary to the Nephele runtime code, the PACT runtime code is loaded dynamically by the Nephele framework and appears as regular user code on this layer. As a result, Nephele has no knowledge about the semantics of the PACT runtime code. Once the PACT runtime code has been invoked, it takes over the control of the task and can only be monitored externally by Nephele, i.e., in terms of its CPU or communication channel utilization.

Through the interface of the respective input contract, the user code which has originally been attached to the PACT task is then executed by the PACT runtime code. Unlike the PACT runtime code, the PACT user code is typically invoked multiple times during the life cycle of a Nephele task. Upon each invocation, the user code is passed a specific subset of the input data according to the respective input contract. The subsets are prepared by the PACT runtime code after the required input data has been retrieved with the help of the Nephele runtime code. Similar to the relation between the Nephele and the PACT runtime code, the PACT runtime code has no knowledge about the

semantics of the user code, expect for the information that can be derived from the optional output contracts.

7.3. Optimization Opportunities through the PACT Layer

After having explained the interaction between code of the Nephele and the PACT layer at runtime, I will now extend the discussion to possible optimization opportunities that arise from exposing PACT semantics to the Nephele layer. In particular, the discussion will focus on requirements for facilitating automated scale-in and scale-out operations at runtime, as described in Section 3.3.

In general, automatic scale-in and scale-out operations, i.e., dynamically adapting the number of running parallel instances, are not possible on the Nephele layer. As described in the previous section, Nephele invokes the user code only once at the beginning of a task's execution and from that point on has neither knowledge of the user code's internal state nor direct control over its behavior. While this design accounts for much of Nephele's flexibility, it renders automatic scaling difficult.

The concrete obstacles for changing the number of Nephele subtasks at runtime lie in Nephele's missing knowledge about the distribution of records among multiple parallel subtasks and the user code's internal state. Adding another parallel instance of a Nephele task at runtime might be incompatible with the distribution strategies employed by the preceding tasks in the DAG. Moreover, eliminating a subtask in order to reduce the task's overall degree of parallelism might corrupt the processing job's result because the subtask's internal state might have been relevant to the further execution.

A simple remedy to this scaling problem on the Nephele layer would be an additional task annotation which declares the encapsulated user code to be either stateful or stateless. In contrast to their stateful counterparts, stateless tasks would not be allowed to buffer any incoming records internally, i.e., any incoming record must be immediately emitted by the user after being processed.

For stateless tasks, the scaling obstacles mentioned above disappear immediately. Since a stateless task is not allowed to buffer any data, it is always safe to destroy parallel instances of it between the processing of two records. Moreover, the data distribution problem becomes irrelevant because a stateless task only forwards records. It never considers records in groups which must potentially obey particular distribution properties.

While general user code would be considered stateful and not to be suitable for automatic scaling, Nephele could instantaneously adapt the degree of parallelism of those tasks which have been declared stateless before. Of course, this approach only solves the

autonomic scaling problem partially. However, it is important to point out that computationally demanding operations, which are the most rewarding targets for automatic scaling anyway, often only process one record at a time and therefore lend themselves well to this kind of annotation. Examples of those computationally complex but stateless operations are the OCR Task and the PDF Creator task from Chapter 4.

Considering the Nephele layer individually, a developer could attach the state annotation to the corresponding user code before submitting the overall Job Graph to Nephele's JobManager. However, in the context of the Stratosphere project, where Nephele mainly acts as a runtime system for the semantically richer PACT layer, it is also conceivable to derive the required state information directly from the PACT program and then make the information available to Nephele at runtime.

While all input contracts of the PACT layer generally stipulate that the encapsulated PACT user code must not preserve any internal state between two consecutive calls, the PACT runtime code, which invokes the user code, must typically maintain some internal state in order to preprocess the incoming key-value pairs according to the respective PACT's input contract. As a result, most of the Nephele tasks which are compiled from the respective vertices of a PACT program are stateful as well.

In general, the PACT runtime code which encapsulates a multi-input contract is always stateful because it has to construct the individual subsets of records from the different input streams, even if no further grouping of the records is required. However, PACT tasks implementing the single-input contracts Map or Reduce can automatically be tagged to be stateless by the PACT compiler in many cases.

The Map input contract represents the simplest case. A PACT task implementing a Map input contract is always stateless, because the PACT runtime code can directly hand any incoming key-value pair to the encapsulated user code. Consequently, the number of parallel instances of the corresponding Nephele task can also in principle be scaled arbitrarily during the job's execution.

The Reduce input contract forces the injected PACT runtime code to group incoming key-value pairs by their key and pass each of those groups to the encapsulated user code separately. In order to construct these groups, the PACT runtime code must initially buffer and sort the incoming key-value pairs before passing them on to the user code. In general, this also makes the resultant Nephele task stateful at runtime. However, in combination with the Unique-Key output contract, the Reduce input contract can also be fulfilled in a stateless manner.

If a developer has attached the Unique-Key output contract to the PACT task preceding the Reduce task in the PACT program DAG, the PACT compiler can assume that all keys of the key-value pairs emitted by the preceding PACT task will be unique during the job's execution. According to this commitment, no parallel instance of the Reduce

task will receive two key-value pairs with identical keys in the course of its execution. Consequently, there is no need for the PACT runtime code to buffer key-value pairs and to construct groups of them for the Reduce user code. The resultant Nephele task becomes stateless and therefore also be suitable for automatic scaling operations.

PACT Input Contract	Properties of Compiled Nephele Task at Runtime
Map	Stateless
Reduce	Stateless after Unique-Key Output Contract, stateful otherwise
Cross	Stateful
Match	Stateful
CoGroup	Stateful

Table 7.1.: State properties of Nephele tasks derived from the encapsulated PACT input contracts.

Table 7.1 summarizes the state properties of Nephele tasks which can automatically be derived either from the encapsulated PACT input contracts or with the help of the attached PACT output contracts. It is worth noticing that the influence of the Unique-Key output contract is not necessarily limited to the state property of a directly connected Reduce task in the overall PACT program. Instead, it may also render successive Reduce tasks in the processing chain stateless given that all intermediate PACT tasks provide either a Same-Key or a Super-Key output contract.

7.4. Summary

This chapter presented the ongoing efforts to enhance massively parallel data analysis on top of IaaS clouds in the scope of the Stratosphere project. Moreover, it explained the relation between the Nephele framework, which has been developed as part of this thesis, and the other components of the Stratosphere software stack, discussed the interaction between those components, and highlighted reasonable first steps for cross-layer optimizations.

In particular, the chapter focused on the PACT layer which is located directly above the Nephele layer in the Stratosphere software stack. In contrast to the Nephele layer, the PACT layer offers a less flexible but more declarative and also semantically richer programming model. I described how the expressiveness of the PACT programming model can be exploited to automatically derive state properties about the compiled Nephele task. By exposing these state properties to the runtime system, for example through annotations to the PACT runtime code, Nephele could take advantage of the additional knowledge and safely adapt the respective task's degree of parallelism during the job's execution.

In a layered software setup like those of the Stratosphere project, propagating information from the specialized, semantically richer layers down to the more general ones is a powerful mechanism to exploit optimization potential. However, these kinds of cross-layer optimizations also create additional dependencies between the components and increase the overall complexity of the system. This carries particular weight for those optimizations opportunities that cannot be derived automatically, for example through a compiler, but must be explicitly pointed out by the developer, for example through annotations like the PACT output contracts, and can potentially lead to wrong processing results.

In sum, I therefore advocate that any efforts for cross-layer optimizations must be clearly traded off against the simplicity of the respective layers' programming abstraction, especially when the programming abstraction of the lower layer is also expected to be used by a developer who has to keep all possible implications of his user code in mind.

8. Conclusion

Infrastructure as a Service clouds play an increasingly important role in the field of massively parallel data processing. While the first approaches in this area were characterized by customer-specific solutions [61], large-scale data analysis has meanwhile moved into the focus of many IaaS platforms and is nowadays offered as a well-established product in the portfolio of major cloud computing companies [14].

However, despite the growing popularity of large-scale data analysis on top of IaaS platforms, the current data processing frameworks, which provide the technical foundation for creating and executing the highly distributed applications, stem from the era of cluster computing and disregard the particular characteristics of the new processing environment. This applies to the *opportunities* that IaaS clouds offer in terms of parallel data processing, namely the on-demand allocation of large sets of heterogeneous computing resources, as well as the *challenges*, i.e., the potentially reduced and less predictable I/O performance compared to classic cluster setups.

This thesis contributes towards improving the efficiency of massively parallel data processing on IaaS clouds by revisiting the design of data processing frameworks for these new platforms. In general, the contributions of the thesis can be found in two major areas, pertaining to the opportunities and the challenges of clouds mentioned above.

The first area of contributions centers around the exploitation of dynamic resource allocation in the context of parallel data processing. Here, for the first time, the thesis presents a series of design principles a data processing framework must meet in order to take advantage of the cloud's new resource provisioning abilities. As a concrete implementation of these design principles, the thesis introduces *Nephele*, the first data processing framework which allows to flexibly adjust the number and type of allocated VMs according to the current processing workload. Unlike existing frameworks, Nephele no longer assumes that it owns the individual computing nodes. Instead, the computing nodes are rather considered to be temporarily leased from an IaaS clouds with a clear notion of monetary cost. The thesis describes the implications of this change of perspective on the programming abstraction and resource scheduling. Moreover, it illustrates different strategies for VM allocation to optimize the processing cost of MTC-like applications.

Motivated by the cloud's pricing model, the thesis also draws attention to the problem of finding reasonable degrees of parallelism for large-scale data analysis applications on

IaaS clouds. As an assistance to the developer, the thesis introduces a novel bottleneck detection scheme for parallel DAG-based data flow programs. The scheme provides the developer with valuable feedback on CPU and I/O bottlenecks that occurred in the course of the job's execution and helps him to successively improve the scale-out of its individual task. The thesis also discusses requirements for automatic scale-in and scale-out operations on the Nephele layer and points out how knowledge from semantically richer programming layers can help to fulfill these.

The second area of contributions addresses the increased risk of I/O bottlenecks a parallel data processing framework faces as a result of the cloud's less predictable I/O characteristics. After having illustrated the potential reasons for I/O degradations, the thesis proposes two different contributions in this area. Both contributions are applicable on an application level and do not require the assistance of the cloud provider.

As the first contribution in this field, the thesis introduces a new adaptive online compression scheme in order to mitigate the negative effects of colocated VMs. Unlike existing adaptive compression schemes, the new approach does not rely on direct system metrics which have been identified to be potentially flawed in virtualized environments. Moreover, the new compression approach does not require an offline training phase, but calibrates itself in the course of the data transmission itself, based on the application data rate. In extensive network experiments the new adaptive compression scheme has been able to improve the throughput of colocated VMs up to a factor of four.

As the second contribution to the cloud's current I/O issues, the thesis proposes a hybrid scheme to infer network topologies which physically interconnect a set of VMs. The scheme thereby accommodates the fact that most parallel data processing frameworks offer to take the physical network topology into account in order to exploit data locality and reduce the risk of I/O bottlenecks, however, this knowledge has not been available to the cloud customer so far. The thesis studies topology inference based on end-to-end measurements in the presence of hardware virtualization and proposes several improvements to increase the inference accuracy for typical data center networks.

Although the thesis has already addressed various research aspects of massively parallel data processing on IaaS platforms, there are several interesting directions for further research in this area. In particular, I want to highlight the following three items:

- **Task colocation and choice of appropriate VM types:** As indicated in Chapter 3, the construction of efficient parallel execution schedules from a submitted Nephele job currently relies heavily on the provided user annotations, especially with regard to the choice of suitable VM types for particular tasks and the colocation of different tasks on the same VM. It is an interesting open research question to what extent a parallel data processing framework could make these decisions for the developer. For example, the data processing framework's scheduler could

attempt to improve the overall utilization of rented cloud resources by mapping a compute and an I/O-intensive task to the same machine. Sensible approaches to this research question could include both feedback learning to construct robust initial execution schedules but also reoptimization of unfavorable scheduling decisions at runtime.

- **Topology-aware scheduling:** With the topology inference service presented in Chapter 6, topology-aware scheduling also becomes a promising field for optimizations. So far, Nephele has been able to map tasks of the overall processing job to arbitrary VMs of a matching type, because without topology information the communication cost between each pair of machines has been the same. In the presence of network topology information, however, it becomes desirable to identify groups of heavily-communicating tasks and schedule the tasks in these groups to run on VMs which are also close to each other in terms of the network topology. Currently, Nephele only applies very simple strategies to detect such groups. Technically, the problem is related to the task mapping problem [32] and has already been discussed in several contexts, such as parallel machines [2], sensor networks [1], or federated databases [137].
- **Fault tolerance:** A subject that has been only discussed marginally by this thesis, but carries special importance in the context of massively parallel data processing is fault tolerance. In a computing setup involving hundreds or even thousands of VMs, individual machines are likely to fail. Current data processing frameworks compensate the risk of resource outages with extensive materializations of intermediate results, even when those intermediate results are easy to recompute or the processing jobs only run for several seconds. A more adaptive approach could help to reduce the current checkpointing overhead by only materializing those intermediate results which are particularly valuable for the job recovery. With regard to the cloud's pricing model, the cost of a VMs in contrast to its reliability represents an interesting tradeoff, for example on the basis of Amazon EC2 spot instances. An advanced fault tolerance scheme for massively parallel data processing could attempt to lower the overall processing cost of a job by balancing the cost savings with those machines' increased probability of failure. First publications have already taken up this idea [149].

Overall, the thesis presents a set of important contributions to the still young research field of massively parallel data processing on IaaS platforms. With the ever increasing amount of data many companies and academic institutions have to deal with, I am confident that cloud computing will continue to attract a growing number of customers as a platform for data-intensive applications and more and more replace the need for classic, dedicated cluster environments.

A. Appendix

Unless explicitly stated otherwise in the respective chapters, all experiments presented in this thesis were conducted on our local IaaS cloud testbed. The testbed consisted of up to 12 physical servers with the following hardware configuration:

CPU	Two Intel Xeon 2.66 GHz CPUs (model E5430)
RAM	32 GB
Hard Disk	Seagate Barracuda ES.2 SATA 500 GB (formatted with ext3 file system)
Network	Intel Corporation 80003ES2LAN 1 GB/s Ethernet

The servers were all connected to a central HP ProCurve 1800-24G switch. The host operating system was Gentoo Linux running the Eucalyptus [99] cloud software.

Since the individual experiments presented in this thesis were conducted over a longer period of time, the concrete versions of the software used may have changed between the experiments. As a result, I will describe the detailed software setup of each experiment separately in the following. However, the cloud testbed was fully dedicated to the experiments during the respective measurements and benchmarks. Thus, side effects from other applications can be excluded in general.

Microbenchmarks in Chapter 2 and Compression Experiments in Chapter 5

For the experiments presented in Chapter 2 and those in Chapter 5 we used Eucalyptus version 1.6. Since we conducted experiments with both KVM and XEN-based VMs, we had to use different versions of the host operating system kernel. For the experiments involving KVM we used the Linux kernel 2.6.32-gentoo-r7. The XEN-based VMs ran on top of a modified XEN Linux kernel with version 2.6.34-xen-r4.

The VMs used during all these experiments had the following characteristics:

CPU	1 CPU core
RAM	2 GB
Hard Disk	60 GB, device driver `scsi` for KVM (full virt.), `virtio_blk` for KVM (paravirt.), and `xenblk` for XEN, formatted with ext2 file system

Network	Bridged network, device driver `e1000` for KVM (full virt.), `virtio_net` for KVM (paravirt.), and `xennet` for XEN
OS	Ubuntu Linux 9.10 (Karmic Koala)
Kernel	2.6.31-22-server for XEN-based VMs, 2.6.32-gentoo-r7 otherwise
Java	Java HotSpot 64-bit Server VM, version 1.6

In order to avoid any side effect during these experiments, we only ran one VM per physical host and shut down all unnecessary background services.

To put the experimental results which we gained on our local Eucalyptus-based cloud into perspective with a commercial cloud system, we also conducted some baseline tests on Amazon EC2. For these tests we instantiated two VMs of type "m1.small" in Amazon's US East (Virginia) data center. Both VMs were created inside the same availability zone and same security group. To reduce the probability of obtaining two VMs which are co-located on the same physical host, we destroyed and reinstantiated the machines several times between different runs of our experiments. For all our experiments on Amazon EC2 we used the VM image "Basic 32-bit Amazon Linux AMI 1.0" with the identifier "ami-08728661". The VMs reported a Linux kernel of version 2.6.34.7-56.40.amzn1.i686. For all experiments which involved file I/O we used the ephemeral storage partition of the VMs with an ext2 file system.

Experiments on Dynamic Resource Provisioning in Chapter 3

In the scope of the experiments as shown in Chapter 3 we only used KVM-based VMs, however, two different types of VMs, i.e., with different hardware characteristics. The version of the host operating system kernel in this case was 2.6.30, Eucalyptus was installed in version 1.5. The concrete configuration of the two VM types we used is given in the following table:

Identifier	m1.small	c1.xlarge
CPU	1 CPU core	8 CPU cores
RAM	1 GB	18 GB
Hard Disk	128 GB	512 GB
	Device driver `virtio_blk`, formatted with ext2 file system	
Network	Bridged network, device driver `virtio_net`	
OS	Ubuntu Linux 9.10 (Karmic Koala)	
Kernel	2.6.28	
Java	Java HotSpot 64-bit Server VM, version 1.6	

Experiments on Bottleneck Detection in Chapter 4

The experiments on bottleneck detection in parallel DAG-based data flow programs were also conducted with KVM-based VMs only. During the experiments the host operating system kernel was of version 2.6.32, Eucalyptus ran in version 1.6. The VM type we used during the experiments had the following configuration:

CPU	1 CPU core
RAM	2 GB
Hard Disk	60 GB, device driver `virtio_blk`, formatted with ext2 file system
Network	Bridged network, device driver `virtio_net`
OS	Ubuntu Linux 9.10 (Karmic Koala)
Kernel	2.6.31
Java	Java HotSpot 64-bit Server VM, version 1.6

In order to mimic a persistent storage service as part of our local IaaS cloud (similar to Amazon EBS), we set up a regular NFS server during the experiments. The server was connected with one GBit/s to the rest of the network.

Experiments on Topology Inference in Chapter 6

Finally, the experiments on topology inference in IaaS clouds again involved KVM as well as XEN-based VMs. As a result, we again set up two different host operating system kernels, version 2.6.34-xen-r4 for XEN-based experiments, 2.6.32-gentoo-r7 for the experiments involving KVM. During the experiments we deployed up to 64 VMs, at most eight VMs per host, using the Eucalyptus cloud software in version 1.6. Each of these VMs had the following configuration:

CPU	1 CPU core
RAM	2 GB
Hard Disk	60 GB, device driver `virtio_blk`, formatted with ext2 file system
Network	Bridged network, device driver `e1000` for KVM (full virt.), `virtio_net` for KVM (paravirt.), and `xennet` for XEN
OS	Ubuntu Linux 9.10 (Karmic Koala)
Kernel	2.6.31-22-server for XEN, 2.6.32-gentoo-r7 otherwise
Java	Java HotSpot 64-bit Server VM, version 1.6

To generate the background traffic during the experiments, we devised a small auxiliary program which was executed on each VM during our experiments. The program was capable of generating UDP traffic at an adjustable data rate. Each generated UDP

packet carried eight KB of payload. To achieve a fair mixture between inter-host and intra-host traffic, we set up four VMs on each physical host to exchange background traffic among each other while the other four VMs transmitted data to VMs on a different host. So, for example, at a background traffic level of 50 MBit/s, all eight VMs generated network traffic at 50 MBit/s. However, only four VMs actually utilized the physical network. Moreover, it is important to point out that the rate at which traffic generation is issued by a user space program is not necessarily equal to the actual data rate observed by the receiver. The actual data rate is determined by the flow control mechanisms employed by the kernel and the virtualization layer.

Bibliography

[1] Zoe Abrams and Jie Liu. Greedy is good: On service tree placement for in-network stream processing. In *Proc. of the 26th IEEE International Conference on Distributed Computing Systems*, ICDCS '06, pages 72–, Washington, DC, USA, 2006. IEEE Computer Society.

[2] Tarun Agarwal, Amit Sharma, and Laxmikant V. Kalé. Topology-aware task mapping for reducing communication contention on large parallel machines. In *Proc. of the 20th International Conference on Parallel and Distributed Processing*, IPDPS '06, pages 145–145, Washington, DC, USA, 2006. IEEE Computer Society.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Computer Communication Review*, 38:63–74, August 2008.

[4] Khaldoon Al-Zoubi and Gabriel Wainer. Using REST web-services architecture for distributed simulation. In *Proc. of the 23rd ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, PADS '09, pages 114–121, Washington, DC, USA, 2009. IEEE Computer Society.

[5] Alexander Alexandrov, Dominic Battré, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. Massively parallel data analysis with PACTs on Nephele. *Proc. of the VLDB Endowment*, 3:1625–1628, September 2010.

[6] Alexander Alexandrov, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. MapReduce and PACT - comparing data parallel programming models. In *Proc. of the 14th Conference on Database Systems for Business, Technology, and Web*, BTW 2011, pages 25–44, Bonn, Germany, 2011. GI.

[7] Amazon Web Services LLC. Amazon EC2 SLA. `http://aws.amazon.com/ec2-sla/`, September 2009.

[8] Amazon Web Services LLC. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`, September 2009.

[9] Amazon Web Services LLC. Amazon Simple Storage Service (Amazon S3). `http://aws.amazon.com/s3/`, September 2009.

[10] Amazon Web Services LLC. Amazon Web Services. `http://aws.amazon.com/`, September 2009.

[11] Amazon Web Services LLC. AWS Import/Export. `http://aws.amazon.com/importexport/`, September 2009.

[12] Amazon Web Services LLC. Elastic Block Storage. `http://aws.amazon.com/ebs/`, September 2009.

[13] Amazon Web Services LLC. Amazon Elastic Compute Cloud. `http://docs.amazonwebservices.com/AWSEC2/latest/APIReference/`, March 2011.

[14] Amazon Web Services LLC. Amazon Elastic MapReduce. `http://aws.amazon.com/de/elasticmapreduce/`, January 2011.

[15] Amazon Web Services LLC. Amazon Web Services Customer Agreement. `http://aws-portal.amazon.com/gp/aws/developer/terms-and-conditions.html`, April 2011.

[16] Amazon Web Services LLC. Atbrox and Lingit case study: Amazon Web Services. `http://aws.amazon.com/solutions/case-studies/atbrox/`, April 2011.

[17] Amazon Web Services LLC. AWS Management Console. `http://aws.amazon.com/console/`, April 2011.

[18] Amazon Web Services LLC. High performance computing. `http://aws.amazon.com/ec2/hpc-applications/`, April 2011.

[19] Amazon Web Services LLC. Razorfish case study: Amazon Web Services. `http://aws.amazon.com/solutions/case-studies/razorfish/`, April 2011.

[20] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53:50–58, April 2010.

[21] Ross Arnold and Tim Bell. A corpus for the evaluation of lossless compression algorithms. In *Proc. of the Data Compression Conference*, DCC '97, pages 201–210, March 1997.

[22] Paul Barford, Azer Bestavros, John Byers, and Mark Crovella. On the marginal utility of network topology measurements. In *Proc. of the 1st ACM SIGCOMM Workshop on Internet Measurement*, IMW '01, pages 5–17, New York, NY, USA, 2001. ACM.

[23] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Operating Systems Review*, 37:164–177, October 2003.

[24] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: A programming model and execution framework for web-scale analytical processing. In *Proc. of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 119–130, New York, NY, USA, 2010. ACM.

[25] Dominic Battré, Natalia Frejnik, Siddhant Goel, Odej Kao, and Daniel Warneke. Evaluation of network topology inference in opaque compute clouds through end-to-end measurements. In *Proc. of the 4th IEEE International Conference on Cloud Computing*, CLOUD '11, pages 17 –24, July 2011.

[26] Dominic Battré, Natalia Frejnik, Siddhant Goel, Odej Kao, and Daniel Warneke. Inferring network topologies in Infrastructure as a Service clouds. In *Proc. of the 11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, CCGRID '11, pages 604–605, May 2011.

[27] Dominic Battré, Matthias Hovestadt, Björn Lohrmann, Alexander Stanik, and Daniel Warneke. Detecting bottlenecks in parallel DAG-based data flow programs. In *Proc. of the 2010 IEEE Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '10, pages 1 –10, November 2010.

[28] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proc. of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[29] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Evaluating the performance of skeleton-based high level parallel programs. In *The International Conference on Computational Science*, ICCS '04, pages 299–306, Heidelberg/Berlin, Germany, 2004. Springer-Verlag GmbH.

[30] Azer Bestavros, John W. Byers, and Khaled A. Harfoush. Inference and labeling of metric-induced network topologies. *IEEE Transactions on Parallel and Distributed Systems*, 16:1053–1065, November 2005.

[31] Kevin S. Beyer, Vuk Ercegovac, Jun Rao, and Eugene J. Shekita. jaql - query language for JavaScript Object Notation (JSON). `http://code.google.com/p/jaql/`, April 2011.

[32] S. H. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, 30:207–214, March 1981.

[33] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proc. of the 27th IEEE International Conference on Data Engineering*, ICDE '11, pages 1151–1162, Washington, DC, USA, 2011. IEEE Computer Society.

[34] Andre Broido and KC Claffy. Internet topology: Connectivity of IP graphs. In *Proc. of the SPIE International Symposium on Convergence of IT and Communication*, SPIE ITCom '01, pages 172–187, Bellingham, WA, USA, August 2001. Society of Photo-Optical Instrumentation Engineers.

[35] Rui M. Castro, Mark J. Coates, and Robert D. Nowak. Likelihood based hierarchical clustering. *IEEE Transactions on Signal Processing*, 52(8):2308–2321, August 2004.

[36] Eduardo César, J.G. Mesa, Joan Sorribes, and Emilio Luque. Modeling master-worker applications in POETRIES. In *Proc. of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 22–30, April 2004.

[37] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. *Proc. of the VLDB Endowment*, 1:1265–1276, August 2008.

[38] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. *SIGOPS Operating Systems Review*, 41:17–30, March 2007.

[39] Ludmila Cherkasova and Rob Gardner. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. In *Proc. of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 24–24, Berkeley, CA, USA, 2005. USENIX Association.

[40] Citrix Systems, Inc. Xen Scheduling. `http://wiki.xensource.com/xenwiki/Scheduling`, April 2011.

[41] Mark Coates, Rui Castro, Robert Nowak, Manik Gadhiok, Ryan King, and Yolanda Tsang. Maximum likelihood network topology identification from edge-based unicast measurements. *SIGMETRICS Performance Evaluation Review*, 30:11–20, June 2002.

[42] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce online. In *Proc. of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI '10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[43] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, January 2008.

[44] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13:219–237, July 2005.

[45] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proc. of the VLDB Endowment*, 3:515–529, September 2010.

[46] Tim Dornemann, Ernst Juhnke, and Bernd Freisleben. On-demand resource provisioning for BPEL workflows using Amazon's Elastic Compute Cloud. In *Proc. of the 9th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, CCGRID '09, pages 140–147, Washington, DC, USA, 2009. IEEE Computer Society.

[47] Alexei Drummond and Korbinian Strimmer. PAL: An object-oriented programming library for molecular evolution and phylogenetics. *Bioinformatics*, 17:662–663, 2001.

[48] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. *SIGOPS Operating Systems Review*, 33:261–276, December 1999.

[49] Nick G. Duffield, Joseph Horowitz, Francesco Lo Presti, and Donald Towsley. Multicast topology inference from end-to-end measurements. *Advances in Performance Analysis*, 3(3):207–226, September 2000.

[50] Nick G. Duffield, Joseph Horowitz, Francesco Lo Presti, and Donald Towsley. Multicast topology inference from measured end-to-end loss. *IEEE Transactions on Information Theory*, 48(1):26 –45, January 2002.

[51] Nick G. Duffield, Francesco Lo Presti, Vern Paxson, and Donald Towsley. Network loss tomography using striped unicast probes. *IEEE/ACM Transactions on Networking*, 14:697–710, August 2006.

[52] Bin Fan, Wittawat Tantisiriroj, Lin Xiao, and Garth Gibson. DiskReduce: RAID for data-intensive scalable computing. In *Proc. of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 6–10, New York, NY, USA, 2009. ACM.

[53] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, Irvine, California, 2000.

[54] Ian T. Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[55] Ian T. Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15:200–222, August 2001.

[56] Ian T. Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. In *Proc. of the Grid Computing Environments Workshop*, GCE '08, pages 1 –10, November 2008.

[57] James Frey, Todd Tannenbaum, Miron Livny, Ian T. Foster, and Steven Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5:237–246, July 2002.

[58] GoGrid, LLC. Cloud Hosting, Cloud Servers, Hybrid Hosting, Cloud Infrastructure from GoGrid, March 2011.

[59] GoGrid, LLC. Service Level Agreement (SLA): GoGird Cloud Hosting. `http://www.gogrid.com/legal/sla.php`, March 2011.

[60] Hector Gonzalez, Jiawei Han, Xiaolei Li, and Diego Klabjan. Warehousing and analyzing massive RFID data sets. In *Proc. of the 22nd International Conference on Data Engineering*, ICDE '06, pages 83–, Washington, DC, USA, 2006. IEEE Computer Society.

[61] Derek Gottfrid. Self-service, Prorated Super Computing Fun! `http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/`, November 2007.

[62] Sriram Govindan, Arjun R. Nath, Amitayu Das, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Xen and co.: Communication-aware CPU scheduling for consolidated Xen-based hosting platforms. In *Proc. of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 126–136, New York, NY, USA, 2007. ACM.

[63] Jim Gray, David T. Liu, Maria Nieto-Santisteban, Alex Szalay, David J. DeWitt, and Gerd Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34:34–41, December 2005.

[64] Robert L. Grossman. The case for cloud computing. *IT Professional*, 11:23–27, March 2009.

[65] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. SOAP version 1.2 part 1: Messaging framework (second edition). `http://www.w3.org/TR/2007/REC-soap12-part1-20070427/`, April 2007.

[66] Ashish Gupta, Marcia Zangrilli, Ananth I. Sundararaj, Anne I. Huang, Peter A. Dinda, and Bruce B. Lowekamp. Free network measurement for adaptive virtualized distributed computing. In *Proc. of the 20th International Conference on Parallel and Distributed Processing*, IPDPS '06, pages 149–149, Washington, DC, USA, 2006. IEEE Computer Society.

[67] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in Xen. In *Proc. of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, Middleware '06, pages 342–362, New York, NY, USA, 2006. Springer-Verlag New York, Inc.

[68] Gary Hardiman. Ultra-high-throughput sequencing, microarray-based genomic selection and pharmacogenomics. *Pharmacogenomics*, 9:5–9, January 2008.

[69] Matthias Hovestadt, Odej Kao, Andreas Kliem, and Daniel Warneke. Evaluating adaptive compression to mitigate the effects of shared I/O in clouds. In *Proc. of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, IPDPSW '11, pages 1042 –1051, May 2011.

[70] Wei Huang, Jiuxing Liu, Bulent Abali, and Dhabaleswar K. Panda. A case for high performance computing with virtual machines. In *Proc. of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 125–134, New York, NY, USA, 2006. ACM.

[71] Shadi Ibrahim, Hai Jin, Lu Lu, Li Qi, Song Wu, and Xuanhua Shi. Evaluating MapReduce on virtual machines: The Hadoop case. In Martin Jaatun, Gansen Zhao, and Chunming Rong, editors, *Cloud Computing*, volume 5931 of *Lecture Notes in Computer Science*, pages 519–528. Springer-Verlag GmbH, Heidelberg/Berlin, Germany, 2009.

[72] IEEE Computer Society. Station and media access control connectivity discovery. `http://standards.ieee.org/getieee802/download/802.1AB-2005.pdf`, December 2010.

[73] Intel Corporation. Intel virtualization technology for directed I/O (VT-d): Enhancing Intel platforms for efficient virtualization of I/O devices. `http://software.intel.com/en-us/articles/intel-virtualization-technology-for-directed-io-vt-d-enhancing-intel-platforms-for-efficient-virtualization-of-io-devices/`, February 2009.

[74] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Operating Systems Review*, 41:59–72, March 2007.

[75] iText Software Corp. Text - Free / Open Source PDF Library for Java and C#. `http://www.itextpdf.com/`, April 2011.

[76] Akshay Java, Xiaodan Song, Tim Finin, and Belle Tseng. Why we twitter: Understanding microblogging usage and communities. In *Proc. of the 9th WebKDD and 1st SNA-KDD 2007 Workshop on Web Mining and Social Network Analysis*, WebKDD/SNA-KDD '07, pages 56–65, New York, NY, USA, 2007. ACM.

[77] Emmanuel Jeannot, Björn Knutsson, and Mats Björkman. Adaptive online data compression. In *Proc. of the 11th IEEE International Symposium on High Performance Distributed Computing*, HPDC '02, pages 379–, Washington, DC, USA, 2002. IEEE Computer Society.

[78] Xing Jin, Wanqing Tu, and S. H. Gary Chan. Scalable and efficient end-to-end network topology inference. *IEEE Transactions on Parallel and Distributed Systens*, 19:837–850, June 2008.

[79] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proc. of the annual conference on USENIX '06 Annual Technical Conference*, pages 1–1, Berkeley, CA, USA, 2006. USENIX Association.

[80] Gueyoung Jung, Galen S. Swint, Jason Parekh, Calton Pu, and Akhil Sahai. Detecting bottleneck in n-tier IT applications through analysis. In *Proc. of the 17th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, DSOM '06, pages 149–160, Heidelberg/Berlin, Germany, 2006. Springer-Verlag GmbH.

[81] Hari Kannan, Mihai Budiu, John D. Davis, and Girish Venkataramani. Tuning SoCs using the global dynamic critical path. In *Proc. of the 2009 IEEE International SOC Conference*, SOCC '09, pages 407 –411, September 2009.

[82] Mukil Kesavan, Ada Gavrilovska, and Karsten Schwan. Differential virtual time (DVT): Rethinking I/O service differentiation for virtual machines. In *Proc. of the 1st ACM Symposium on Cloud computing*, SoCC '10, pages 27–38, New York, NY, USA, 2010. ACM.

[83] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: The Linux virtual machine monitor. In *Proc. of the 2007 Ottawa Linux Symposium*, OLS '07, pages 225–230, July 2007.

[84] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir performance analysis tool-set. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 139–155. Springer-Verlag GmbH, Heidelberg/Berlin, Germany, 2008.

[85] Donald Kossmann, Tim Kraska, and Simon Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *Proc. of the 2010 International Conference on Management of Data*, SIGMOD '10, pages 579–590, New York, NY, USA, 2010. ACM.

[86] Michael A. Kozuch, Michael P. Ryan, Richard Gass, Steven W. Schlosser, David O'Hallaron, James Cipar, Elie Krevat, Julio López, Michael Stroucken, and Gregory R. Ganger. Tashi: Location-aware cluster management. In *Proc. of the 1st Workshop on Automated Control for Datacenters and Clouds*, ACDC '09, pages 43–48, New York, NY, USA, 2009. ACM.

[87] Chandra Krintz and Sezgin Sucu. Adaptive on-the-fly compression. *IEEE Transactions on Parallel and Distributed Systems*, 17:15–24, January 2006.

[88] Li Li and Allen D. Malony. Model-based performance diagnosis of master-worker parallel computations. In *Proc. of the 12th International Euro-Par Conference*, Euro-Par '06, pages 35–46, Heidelberg/Berlin, Germany, 2006. Springer-Verlag GmbH.

[89] Robert L. Mack, Sougata Mukherjea, Aya C. Soffer, Naohiko Uramoto, Erik W. Brown, Anni R. Coden, James W. Cooper, Akihiro Inokuchi, Balakrishna R. Iyer, Yosi Mass, Hirofumi Matsuzawa, and L. Venkata Subramaniam. Text analytics for life science using the unstructured information management architecture. *IBM Systems Journal*, 43:490–515, July 2004.

[90] Simon Malkowski, Markus Hedwig, Deepal Jayasinghe, Calton Pu, and Dirk Neumann. CloudXplor: A tool for configuration planning in clouds based on empirical data. In *Proc. of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 391–398, New York, NY, USA, 2010. ACM.

[91] Simon Malkowski, Markus Hedwig, Jason Parekh, Calton Pu, and Akhil Sahai. Bottleneck detection using statistical intervention analysis. In *Proc. of the Distributed Systems: Operations and Management 18th IFIP/IEEE International Conference on Managing Virtualization of Networks and Services*, DSOM '07, pages 122–134, Heidelberg/Berlin, Germany, 2007. Springer-Verlag GmbH.

[92] Paul Marshall, Kate Keahey, and Tim Freeman. Elastic site: Using clouds to elastically extend site resources. In *Proc. of the 10th IEEE/ACM International Conference on Cluster, Cloud, and Grid Computing*, CCGRID '10, pages 43–52, Washington, DC, USA, 2010. IEEE Computer Society.

[93] Peter Mell and Tim Grance. The NIST definition of cloud computing (v15). Technical report, National Institute of Standards and Technology, 2009.

[94] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *Proc. of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 13–23, New York, NY, USA, 2005. ACM.

[95] Nitin Motgi and Amar Mukherjee. Network conscious text compression system (NCTCSys). In *Proc. of the International Conference on Information Technology: Coding and Computing*, pages 440–446, Washington, DC, USA, 2001. IEEE Computer Society.

[96] Gavin Mulligan and Denis Gracanin. A comparison of SOAP and REST implementations of a service based interaction independence middleware framework. In *Proc. of the 2009 Winter Simulation Conference*, WSC '09, pages 1423–1432, December 2009.

[97] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *Proc. of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association.

[98] Jian Ni, Haiyong Xie, Sekhar Tatikonda, and Yang Richard Yang. Efficient and dynamic routing topology inference from end-to-end measurements. *IEEE/ACM Transactions on Networking*, 18:123–135, February 2010.

[99] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus open-source cloud-computing system. In *Proc. of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[100] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proc. of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[101] Owen O'Malley and Arun C. Murthy. Winning a 60 second dash with a yellow elephant. Technical report, Yahoo!, 2009.

[102] Simon Ostermann, Alexandria Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. A performance analysis of EC2 cloud computing services for scientific computing. In Ozgur Akan, Paolo Bellavista, Jiannong Cao, Falko Dressler, Domenico Ferrari, Mario Gerla, Hisashi Kobayashi, Sergio Palazzo, Sartaj Sahni, Xuemin (Sherman) Shen, Mircea Stan, Jia Xiaohua, Albert Zomaya, Geoffrey Coulson, Dimiter R. Avresky, Michel Diaz, Arndt Bode, Bruno Ciciani, and Eliezer Dekel, editors, *Cloud Computing*, volume 34 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 115–131. Springer-Verlag GmbH, Heidelberg/Berlin, Germany, 2010.

[103] Mayur R. Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon S3 for science grids: A viable solution? In *Proc. of the 2008 International Workshop on Data-aware Distributed Computing*, DADC '08, pages 55–64, New York, NY, USA, 2008. ACM.

[104] Igor Pavlow. LZMA SDK (software Development Kit). `http://www.7-zip.org/sdk.html`, January 2011.

[105] Rackspace US, Inc. The Rackspace Cloud. `http://www.rackspacecloud.com/`, December 2010.

[106] Ioan Raicu, Ian T. Foster, and Yong Zhao. Many-task computing for grids and supercomputers. In *Proc. of the 1st Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '08, pages 1 –11, November 2008.

[107] Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian T. Foster, and Michael Wilde. Falkon: A Fast and Light-weight tasK executiON framework. In *Proc. of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 1 –12, November 2007.

[108] Lavanya Ramakrishnan, Charles Koelbel, Yang-Suk Kee, Rich Wolski, Daniel Nurmi, Dennis Gannon, Graziano Obertelli, Asim YarKhan, Anirban Mandal, T. Mark Huang, Kiran Thyagaraja, and Dmitrii Zagorodnov. VGrADS: Enabling e-science workflows on grids and clouds with fault tolerance. In *Proc. of the 2009 ACM/IEEE Conference on Supercomputing*, SC '09, pages 47:1–47:12, New York, NY, USA, 2009. ACM.

[109] Sylvia Ratnasamy and Steven McCanne. Inference of multicast routing trees and bottleneck bandwidths using end-to-end measurements. In *Proc. of the 18th Conference on Information Communications*, volume 1 of *INFOCOM '99*, pages 353–360, March 1999.

[110] Lasse Mikkel Reinhold. Fast compression library for C, C# and Java. `http://www.quicklz.com/`, January 2011.

[111] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proc. of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[112] D. F. Robinson and L. R. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1-2):131 – 147, 1981.

[113] Rusty Russell. virtio: Towards a de-facto standard for virtual I/O devices. *SIGOPS Operating Systems Review*, 42:95–103, July 2008.

[114] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proc. of the VLDB Endowment*, 3:460–471, September 2010.

[115] Seetharami R. Seelam and Patricia J. Teller. Virtual I/O scheduler: A scheduler of schedulers for performance virtualization. In *Proc. of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 105–115, New York, NY, USA, 2007. ACM.

[116] Jeffrey Shafer. I/O virtualization bottlenecks in cloud computing today. In *Proc. of the 2nd Conference on I/O Virtualization*, WIOV '10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[117] Sameer S. Shende and Allen D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20:287–311, May 2006.

[118] Meng-Fu Shih and Alfred O. Hero. Hierarchical inference of unicast network topologies based on end-to-end measurements. *IEEE Transactions on Signal Processing*, 55(5):1708–1718, May 2007.

[119] Tatsuya Shirai, Hideo Saito, and Kenjiro Taura. A fast topology inference: A building block for network-aware parallel processing. In *Proc. of the 16th International Symposium on High Performance Distributed Computing*, HPDC '07, pages 11–22, New York, NY, USA, 2007. ACM.

[120] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's LEarning Optimizer. In *Proc. of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 19–28, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[121] Jaspar Subhlok and Gary Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *Proc. of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '96, pages 62–71, New York, NY, USA, 1996. ACM.

[122] Andrew S. Tanenbaum. Network protocols. *ACM Computing Surveys*, 13:453–489, December 1981.

[123] The Apache Software Foundation. MapReduce tutorial. `http://hadoop.apache.org/mapreduce/docs/current/mapred_tutorial.html`, April 2011.

[124] The Apache Software Foundation. Welcome to Apache Hadoop! `http://hadoop.apache.org/`, January 2011.

[125] The Cascading Project. Cascading. `http://www.cascading.org/`, April 2011.

[126] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. of the VLDB Endowment*, 2:1626–1629, August 2009.

[127] Omesh Tickoo, Ravi Iyer, Ramesh Illikkal, and Don Newell. Modeling virtual machine performance: Challenges and approaches. *SIGMETRICS Performance Evaluation Review*, 37:55–60, January 2010.

[128] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf. `http://iperf.sourceforge.net/`, April 2011.

[129] Yolanda Tsang, Mehmet Yildiz, Paul Barford, and Robert Nowak. On the performance of round trip time network tomography. In *Proc. of the 2006 IEEE Internation Conference on Communications*, volume 2 of *ICC '06*, pages 483–488, June 2006.

[130] Yehuda Vardi. Network tomography: Estimating source-destination traffic intensities from link data. *Journal of the American Statistical Association*, 91(433):365–377, March 1996.

[131] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea, and Seth C. Goldstein. Global critical path: A tool for system-level timing analysis. In *Proc. of the 44th ACM/IEEE Design Automation Conference*, DAC '07, pages 783–786, June 2007.

[132] VMware, Inc. Performance evaluation of Intel EPT hardware assist. `http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf`, December 2009.

[133] Gregor von Laszewski, Mihael Hategan, and Depti Kodeboyina. *Workflows for E-science: Scientific Workflows for Grids*, chapter Grid Workflows, pages 340–356. Springer-Verlag GmbH, Heidelberg/Berlin, Germany, 2007.

[134] Nagavijayalakshmi Vydyanathan, Umit Catalyurek, Tahsin Kurc, Ponnuswamy Sadayappan, and Joel Saltz. A duplication based algorithm for optimizing latency under throughput constraints for streaming workflows. In *Proc. of the 37th International Conference on Parallel Processing*, ICPP '08, pages 254–261, Washington, DC, USA, 2008. IEEE Computer Society.

[135] Edward Walker. Benchmarking Amazon EC2 for high-performance scientific computing. *USENIX; login: magazine*, 33(5):18–23, October 2008.

[136] Guohui Wang and T. S. Eugene Ng. The impact of virtualization on network performance of Amazon EC2 data center. In *Proc. of the 29th Conference on Information Communications*, INFOCOM '10, pages 1163–1171, Piscataway, NJ, USA, 2010. IEEE Press.

[137] Xiaodan Wang, Randal C. Burns, Andreas Terzis, and Amol Deshpande. Network-aware join processing in global-scale database federations. In *Proc. of the 24nd International Conference on Data Engineering*, ICDE '08, pages 586 –595, Washington, DC, USA, April 2008. IEEE Computer Society.

[138] Daniel Warneke and Odej Kao. Nephele: Efficient parallel data processing in the cloud. In *Proc. of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, pages 8:1–8:10, New York, NY, USA, 2009. ACM.

[139] Daniel Warneke and Odej Kao. Exploiting dynamic resource allocation for efficient parallel data processing in the cloud. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):985–997, June 2011.

[140] David Wentzlaff, Charles Gruenwald, III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An operating system for multicore and clouds: Mechanisms and implementation. In *Proc. of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 3–14, New York, NY, USA, 2010. ACM.

[141] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 1st edition, June 2009.

[142] Yair Wiseman, Karsten Schwan, and Patrick Widener. Efficient end to end data exchange using configurable compression. *SIGOPS Operating Systems Review*, 39:4–23, July 2005.

[143] Felix Wolf and Bernd Mohr. KOJAK - A tool set for automatic performance analysis of parallel applications. In *Proc. of the 9th International Euro-Par Conference*, volume 2790 of *Euro-Par '09*, pages 1301–1304, Heidelberg/Berlin, Germany, August 2003. Springer-Verlag GmbH. Demonstrations of Parallel and Distributed Computing.

[144] Timothy Wood, Ludmila Cherkasova, Kivanc Ozonat, and Prashant Shenoy. Profiling and modeling resource usage of virtualized applications. In *Proc. of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 366–387, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[145] Xianghua Xu, Feng Zhou, Jian Wan, and Yucheng Jiang. Quantifying performance properties of virtual machine. In *Proc. of the International Symposium on Information Science and Engineering*, volume 1 of *ISISE '08*, pages 24 –28, December 2008.

[146] Hiroshi Yamada and Kenji Kono. Foxytechnique: Tricking operating system policies with a virtual machine monitor. In *Proc. of the 3rd international conference on Virtual execution environments*, VEE '07, pages 55–64, New York, NY, USA, 2007. ACM.

[147] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: Simplified relational data processing on large clusters. In *Proc. of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1029–1040, New York, NY, USA, 2007. ACM.

[148] Bin Yao, Ramesh Viswanathan, Fangzhe Chang, and Daniel Waddington. Topology inference in the presence of anonymous routers. In *Proc. of the 22th Conference on Information Communications*, volume 1 of *INFOCOM '03*, pages 353–363, March 2003.

[149] Sangho Yi, Derrick Kondo, and Artur Andrzejak. Reducing costs of spot instances via checkpointing in the Amazon Elastic Compute Cloud. In *Proc. of the 3rd IEEE International Conference on Cloud Computing*, CLOUD '10, pages 236–243, Washington, DC, USA, July 2010. IEEE Computer Society.

[150] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[151] Yong Zhao, Mihael Hategan, Ben Clifford, Ian T. Foster, Gregor von Laszewski, Veronika Nefedova, Ioan. Raicu, Tiberiu Stef-Praun, and Michael Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Proc. of the 2007 IEEE Congress on Services*, pages 199 –206, July 2007.

[152] Zimory GmbH. Zimory GmbH: HOME. `http://www.zimory.com/`, April 2011.